Construção de uma RPU (reconfigurable processing unit) para algoritmos genéticos baseada em computação reconfigurável

Priscila Aparecida de Moraes

Orientador: *Prof. Dr. Eduardo Marques*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Matemática.

“VERSÃO REVISADA APÓS A DEFESA”

Data da Defesa:	15/04/2004
Visto do Orientador:

USP – São Carlos
Novembro/2002

25/11/02

Concedei-nos, Senhor, a necessária
SERENIDADE para aceitar as coisas que não podemos modificar.
CORAGEM para modificar as que podemos, e
SABEDORIA para distinguirmos umas das outras.

Agradecimentos

Para que eu pudesse agradecer a todas as pessoas que foram importantes para mim nesses anos de graduação e pós, seria necessário uma nova dissertação. Fica então aqui, implícito o meu agradecimento a todas as pessoas marcaram de alguma forma a minha vida nesses 6 anos, e sei que essas pessoas sabem que marcaram.

Primeiramente, gostaria de agradecer a Deus por ter me dado tudo na vida. Por ter colocado no meu caminho pessoas maravilhosas que me amam e apoiam. Agradeço por Ele ter me dado o livre arbítrio que me permitiu cair, e por estar lá embaixo esperando para poder me carregar nos braços quando sabia que eu não podia andar sozinha.

Aos meus pais, João e Vilma, que são tudo na minha vida. Agradeço pelo apoio e amor incondicional que me oferecem. Por dedicarem suas vidas a mim e minha irmã, por suas orações, por todos os sacrifícios, enfim por tudo que fizeram por mim. Sem eles, nada seria possível;

Ao meu orientador, Prof. Dr. Eduardo Marques por ser essa pessoa maravilhosa que me ajuda em tudo que preciso. Agradeço pelo apoio que tem me dado na caminhada da vida, pelos conselhos, pela preocupação, enfim por ser meu amigo além de me orientar no mestrado;

Ao Prof. Dr. Eduardo do Valle Simões que fez um papel muito importante no decorrer dessa dissertação. Por sempre ter arrumado um tempo para me ajudar e ter sido fundamental para a conclusão desse mestrado;

A minha melhor amiga, Aline, pelos dez anos de amizade, porque sei que apesar da distância que nos separa, está e sempre estará ao meu lado me apoiando e dando bronca nas horas necessárias;

A minha irmã, Tassiane, por ser muito mais do que uma irmã, por ser a melhor irmã do mundo. E por ter cuidado de mim quando eu estava mal;

As minhas amigas fiéis, Juliana Garrido, Viviane e Juliana Cobre, que me receberam de volta com os braços abertos. Que me deram o apoio fundamental quando eu mais precisei. Agradeço pelos quatro anos maravilhosos que moramos juntas. Sentirei saudades, mas quero que saibam que mesmo distante estaremos juntas para sempre;

Aos meus companheiros do LCR, Menotti, Vanderlei, Jean, Leandro, Marcão, Adriano, Daniel e Henrique, que alegraram o ambiente de trabalho e me ajudaram sempre que possível. A eles, que as vezes podem até pensar que não influenciaram na conclusão desse trabalho, mas foram de fundamental importância;

As meninas que moram comigo, Fabiana, Aline, Tatiani e Kattiana pela ajuda prestada nessa reta final.

A pessoas muito especiais do ICMC que são:

A Marilia Finazzi de A. Marino, minha amiga de todas as horas;

Profa. Dra. Solange Rezende, pelo carinho e pelos conselhos profissionais e pessoais que me ajudaram muito a crescer;

Profa. Dra. Maria Carolina Monard, uma pessoa excepcional em todos os sentidos, pelo carinho e preocupação;

As meninas da seção de pós-graduação Beth, Laura e Ana Paula que sempre trabalharam com dedicação e amor para que todas as coisas dessem certo;

A Silvana, ao Zé Angelo e a Fernanda Ortega porque de alguma forma foram marcantes para mim.

A todos os meus professores da graduação e pós, porque me ensinaram que só o que se aprende em sala de aula não é suficiente.

A CAPES pelo apoio financeiro.

Abstract

This work consists a FPGA (Field-Programmable Gate Array) implementation of a Reconfigurable Processing Unit (RPU) containing Genetic Algorithm operators of selection, crossover and mutation. The goal of this work is to make available to circuit designers a non-processed module of a genetic algorithm. This RPU is included in a SOPC (System-On-a-Programmable-Chip) designed to implement a navigation controller to a population of autonomous mobile robots. The robot should learn to navigate and explore the environment, avoiding obstacles on its way. The results compare the embedded GA performance for various combinations of mutation and crossover parameters and population sizes.

Resumo

Esse trabalho consiste na implementação em hardware (FPGA) de uma Unidade de Processamento Reconfigurável (RPU) para Algoritmos Genéticos (AGs), com o intuito de disponibilizar um módulo de AG não processado para projetistas de circuitos integrados. Para o desenvolvimento do projeto, foi utilizada codificação binária dos cromossomos, elitismo como método de seleção, no qual o melhor indivíduo não somente é preservado, mas também recombinado com todos os outros. Além disso, também foram utilizados os operadores normais de crossover e mutação. Para os testes foram variados as taxas de crossover e mutação, além do tamanho da população. Os resultados foram obtidos em simulação e corresponderam as expectativas. Essa unidade ainda será implantada num robô para que ele possa evoluir a seu controlador de navegação. O robô aprenderá sozinho a navegar em um ambiente desconhecido, procurando explorar sem colidir com os obstáculos.

Sumário

Lista de Figuras	xi
Lista de Tabelas	xii
Lista de Abreviaturas	xii
1 Introdução, Justificativa e Objetivos	15
2 Algoritmos Genéticos	19
2.1 Introdução	19
2.2 Objetivos dos Algoritmos Genéticos	20
2.3 As Terminologias Biológicas	21
2.4 Esquemas	23
2.5 Métodos de Seleção	26
2.5.1 Roda da Roleta	26
2.5.2 Elitismo	27
2.6 Operadores nos Algoritmos Genéticos	27
2.6.1 Crossover	28
2.6.1.1 Crossover de um ponto	28
2.6.1.2 Crossover de N pontos e Uniforme	29
2.6.2 Mutação	30
2.7 Aplicações	31
2.8 Implementações de Algoritmos Genéticos	32
2.8.1 Os Algoritmos Genéticos na Robótica	32

2.8.1.1	Definição do Problema	33
2.8.2	Implementações em Hardware de Algoritmos Genéticos	36
2.8.2.1	Hardware-Based Genetic Algorithm (HGA)	36
2.8.2.2	Os Módulos e as Funções	36
2.8.2.3	Comparação com um AG em software	38
3	Conceitos Básicos	39
3.1	Modelagem de Hardware	39
3.1.1	Domínios e Níveis de Modelamento	40
3.1.2	Linguagens de Modelamento	43
3.2	Computação Reconfigurável	44
3.2.1	Field-Programmable Gate Arrays (FPGAs)	45
3.2.2	Família Stratix	46
3.2.2.1	Memória	46
3.2.2.2	Blocos de DSP (<i>Digital Signal Processing</i>)	47
3.2.2.3	Dispositivos de Entrada e Saída	48
3.2.2.4	<i>Logic Array Blocks - LABs</i>	49
3.2.3	Hardware Virtual	49
3.2.4	Sistemas Reconfiguráveis	52
3.2.5	Perspectivas da Computação Reconfigurável	52
4	Descrição do Sistema ARCHITECT-R	54
4.1	Hardware do Sistema ARCHITECT-R	55
4.2	Diagrama de Blocos do Sistema ARCHITECT-R	55
4.3	LEGUS System	57
4.4	Considerações Finais	58
5	Implementação do AG	59
5.1	Informações Adicionais	59
5.1.1	Simulador de Linguagens de Descrição de Hardware ModelSim	59
5.2	Semiconductor Reuse Standard - SRS	60
5.3	Desenvolvimento	62
5.3.1	AG utilizando Roda da Roleta Modificado	62
5.3.1.1	Representação	63

5.3.1.2	Funcionamento do Sistema	64
5.3.1.3	Resultados	65
5.3.2	Utilizando Elitismo	65
5.3.2.1	Representação	66
5.3.2.2	Funcionamento do Sistema	66
5.3.2.3	Resultados	66
5.3.3	Módulos Utilizados	67
5.3.3.1	Gerador de Números Aleatórios	67
5.3.3.2	Resultados	68
5.3.3.3	Crossover	69
5.3.3.4	Mutação	70
5.3.3.5	Avaliador	71
5.3.3.6	Alimentador	71
5.4	Testes	72
6	Resultados e Conclusão	73
6.1	Experimentos	73
6.2	Conclusão	77
6.3	Trabalhos Futuros	78
	Referências Bibliográficas	79
A	Gráficos	i

Lista de Figuras

1.1	Objetivos do projeto ARCHITECT-R [Gonçalves, 2001]	18
2.1	Esquema de um Algoritmo Genético simples.	22
2.2	Esquemas [Carvalho and Lacerda., 1999].	24
2.3	Representação dos esquemas de $O(2)$ [Koza, 1992].	25
2.4	Representação dos esquemas de $O(1)$ [Koza, 1992].	25
2.5	Desempenho do AG com e sem elitismo [Carvalho and Lacerda., 1999].	28
2.6	Crossover de um ponto.	28
2.7	Crossover de dois pontos.	29
2.8	Crossover de quatro pontos.	30
2.9	Crossover de dois pontos visto como um anel [Carvalho and Lacerda., 1999].	30
2.10	Crossover Uniforme.	30
2.11	Mutação.	31
2.12	Representação binária do caminho [Sugihara and Smith, 1997]	34
2.13	Exemplo de caminhos encontrados utilizando AG [Sugihara and Smith, 1997]	35
2.14	Exemplo de um terreno em Zig-Zag [Sugihara and Smith, 1996b]	35
2.15	Esquemático do HGA [Scott et al., 1995].	37
3.1	O Diagrama de Gajski-Kuhn, ou Diagrama Y [Calazans, 1995].	42
3.2	Vantagens da computação reconfigurável [Ienne et al., 2002].	45
3.3	Diagrama de blocos dos FPGAs da família Stratix [Altera, 2003a].	46
3.4	Tecnologia TriMatrix da Altera [Altera, 2003a].	47

3.5	Estrutura geral dos blocos de DSP [Altera, 2003a]	48
3.6	Estrutura LAB da Stratix [Altera, 2003a].	50
3.7	Formas de acoplamento de um hardware reconfigurável [Compton and Hauck, 1999].	50
3.8	Formas de reconfiguração [Compton and Hauck, 1999].	51
4.1	Diagrama de blocos do ARCHITECT-R [Gonçalves, 2001].	56
4.2	Diagrama de blocos do <i>Legus System</i> [Gonçalves, 2002].	58
5.1	Cabeçalho do padrão SRS.	61
5.2	Cromossomo utilizado para a codificação dos indivíduos.	64
5.3	Gerador de Números aleatórios de 16 bits.	68
5.4	Registradores da máscara da mutação.	70
5.5	Exemplo de conexões para a máscara se o cromossomo fosse de 4 bits.	70
6.1	Resultado para uma população de 10 indivíduos com 4,34% para várias taxas de crossover entre 0 e 100%.	76
6.2	Resultado para uma população de 50 indivíduos com 6,562% para várias taxas de crossover entre 0 e 100%.	76
6.3	Resultado para uma população de 256 indivíduos com 15,23% para várias taxas de crossover entre 0 e 100%.	77
A.1	Resultados com População de 10 indivíduos e Mutação de 4,34%.	i
A.2	Resultados com População de 10 indivíduos e Mutação de 6,65%.	ii
A.3	Resultados com População de 10 indivíduos e Mutação de 12,31%.	ii
A.4	Resultados com População de 10 indivíduos e Mutação de 15,23%.	ii
A.5	Resultados com População de 50 indivíduos e Mutação de 4,34%.	iii
A.6	Resultados com População de 50 indivíduos e Mutação de 6,65%.	iii
A.7	Resultados com População de 50 indivíduos e Mutação de 12,31%.	iii
A.8	Resultados com População de 50 indivíduos e Mutação de 15,23%.	iv
A.9	Resultados com População de 256 indivíduos e Mutação de 4,34%.	iv
A.10	Resultados com População de 256 indivíduos e Mutação de 6,65%.	iv
A.11	Resultados com População de 256 indivíduos e Mutação de 12,31%.	v
A.12	Resultados com População de 256 indivíduos e Mutação de 15,23%.	v

Lista de Tabelas

2.1	Tabela comparativa entre a ordem do esquema e sua representação geométrica.	25
2.2	Performance do HGA e do SGA [Scott et al., 1995].	38
3.1	Quantidade de recursos dos dispositivos da família Stratix [Altera, 2003a]. . . .	48
5.1	Resultados obtidos no dispositivo EP1S10F780C5ES da família Stratix.	65
5.2	Resultados obtidos no dispositivo EP1S10F780C5ES da família Stratix.	67
5.3	Tabela das conexões do gerador de números aleatórios.[Alfke, 1996]	68
5.4	Arquivo de Saída do teste do Bloco de Gerador de Números Aleatórios de 4 bits No arquivo, a primeira coluna corresponde aos primeiros números gerados e assim por diante.. . . .	69
5.5	Tabela verdade de um bit utilizada para gerar a expressão lógica do Crossover. .	70
6.1	Tempo em ms que o hardware utiliza para encontrar o melhor indivíduo em uma população de 10 indivíduos.	74
6.2	Tempo em ms que o hardware utiliza para encontrar o melhor indivíduo em uma população de 50 indivíduos.	74
6.3	Tempo em ms que o hardware utiliza para encontrar o melhor indivíduo em uma população de 256 indivíduos.	75

Lista de Abreviaturas

AG - Algoritmos Genéticos
ASIC - Application Specific Integrated Circuit
CES - C for Embedded Systems
DDR - Double Data Rate
DNA - Deoxyribonucleic Acid
DSP - Digital Signal Processing
EAB - Embedded Array Blocks
ESB - Embedded System Blocks
FCRAM - Fast Cycle RAM
FIR - Finite Impulse Response
FPGA - Field Programmable Gate Array
FU - Functional Unit
GNU - GNU's not UNIX
HGA - Hardware-Based Genetic Algorithm
IIR - Infinite Impulse Response
IP - Intellectual Property
I/O - In/Out
IOE - I/O Element
ILP - Instruction-Level Parallelism
LAB - Logic Array Blocks
LE - Logic Elements

LISP - List Processing language
LUT - Look-Up Table
PLD - Programmable Logic Device
QDR - Quad Data Rate
RISC - Reduced Instruction Set Computing
RPU - Reconfigurable Processing Unit
RTL - Register-Transfer Level
SDRAM - Synchronous Dynamic Random Access Memory
SOPC - System on a Programmable Chip
SRS - Semiconductor Reuse Standard
VC - Virtual Component
VHDL - VHSIC Hardware Description Language
VHSIC - Very High Speed Integrated Circuit
ZBT - Zero Bus Turnaround

Introdução, Justificativa e Objetivos

Uma área que tem sido muito pesquisada nos dias atuais é a robótica. Muitos pesquisadores têm desenvolvido sistemas para controlar principalmente a navegação dos mesmos [Simões, 2000] [Duckett, 2002]. Além disso, a indústria de desenvolvimento de hardware tem crescido muito nos últimos anos e espera-se que esse crescimento se torne ainda maior. Isso se deve principalmente ao fato de que atualmente é rara a produção de alguns eletrodomésticos, automóveis, etc., que não possuam algum tipo de circuito integrado embutido. Algumas empresas calculam que o número de circuitos integrados que estarão presentes nesses produtos seja cada vez maior e estão fazendo profundos investimentos na área, como é o caso da HP [HP, 2002].

Para satisfazer a essa demanda as empresas necessitam (e necessitarão cada vez mais) de funcionários altamente capacitados nas áreas de pesquisa e desenvolvimento de hardware. Infelizmente é difícil conseguir esse tipo de profissional: quer seja pela alta demanda, quer seja pela falta de uma política de educação adequada para sua formação.

Devido a essas dificuldades, as empresas estão investindo em pesquisas que visam automatizar o processo de criação de circuitos integrados. Existem duas linhas principais de pesquisas nesse sentido: 1) geradores automáticos de hardware à partir de códigos em linguagens de programação de alto nível e 2) Eletrônica Evolutiva.

Os geradores automáticos de hardware são softwares que criam hardwares dedicados a de-

sempenhar determinadas tarefas a partir de alguns parâmetros passados pelos usuários. Como se baseiam na aplicação, podem ser construídos geradores de hardware para AGs, redes neurais, suporte *vector machine*, lógica *fuzzy*, etc. Seu desenvolvimento não é tão simples, mas sua utilização sim: o desenvolvedor do robô necessita apenas definir a aplicação e os parâmetros mais apropriados a ela e, em seguida, integrar o hardware dedicado com o restante do hardware e software do robô. Essa categoria parte do pressuposto de que desenvolver software é mais simples do que desenvolver hardware. Pesquisas nessa área exigem profissionais bastante qualificados nas áreas de projeto de hardware, projeto de compiladores, inteligência artificial, arquitetura de computadores, sistemas distribuídos, entre outras. Existem inúmeras pesquisas desse tipo sendo desenvolvidas tanto por empresas [Rau and Schlansker, 2001] [Faraboschi et al., 2000], quanto por universidades [Cardoso and Vestias, 2003] [Razdan and Smith, 1994] [Wirthlin and Hutchings, 1995] [Hauck et al., 1997] [Goldstein et al., 2000] [Hauser and Wawrzynek, 1997]. Os resultados obtidos têm sido bastante satisfatórios. O maior problema da área de geradores automáticos de hardware é o de se encontrar profissionais que mesclam conhecimento em todas essas áreas tão distintas.

A segunda linha de pesquisa, Eletrônica Evolutiva, reduz o problema de desenvolvimento de hardware a um problema de busca, aplicando Algoritmos Evolutivos para efetuá-las [Zebulum et al., 2002]. Um tipo de Algoritmo Evolutivo, o que foi utilizado nesse trabalho, é chamado de Algoritmo Genético. Uma das maiores preocupações quanto ao uso de Algoritmos Genéticos na robótica diz respeito ao seu desempenho: muitas vezes os AGs obtêm a resposta desejada, mas o tempo que eles levam para obtê-la não pode ser tolerado pela aplicação. Uma possível solução para esse problema seria construir um hardware dedicado para esse fim. No entanto, o desenvolvimento desse hardware seria bastante complexo e tomaria um tempo bastante grande do projetista, o qual precisaria ter bons conhecimentos de AGs, desenvolvimento de hardware e do projeto do robô para o qual o hardware estaria sendo desenvolvido. Além disso, cada vez que um novo robô que utilizasse AGs fosse criado seria necessária a construção de um novo hardware que fosse específico para aquele robô. Outro problema que poderia surgir é a descoberta de uma outra implementação de Algoritmos Genéticos mais eficiente para aquele robô. Nesse caso o projetista do robô deveria decidir entre criar um novo hardware (o que é caro e trabalhoso) ou manter o hardware atual (o que possui um custo em termos de eficiência do robô). Isso pode ser resolvido através da utilização da computação reconfigurável [DeHon and Wawrzynek, 1999], a qual permite que o hardware que está sendo utilizado pelo

robô possa ser trocado a qualquer momento (inclusive com o robô em operação).

Com relação ao Gerador de Hardware, pode-se dizer que a escolha dos parâmetros a serem passados e a integração do hardware gerado com o restante do sistema do robô não é uma tarefa muito fácil. Contudo, ela poderia ser facilitada através do uso de um sistema que gere e integre automaticamente o hardware dedicado ao robô. Um sistema que possui essa finalidade, denominado ARCHITECT-R está sendo desenvolvido no LCR (Laboratório de Computação Reconfigurável) do ICMC (Instituto de Ciências Matemáticas e de Computação) da USP (Universidade de São Paulo) [Gonçalves et al., 2000] [Gonçalves et al., 2001] [Gonçalves, 2001].

O objetivo do Projeto ARCHITECT-R (Fig. 1.1) é tornar-se uma ferramenta avançada de desenvolvimento de hardware para robótica. A partir de uma dada aplicação, escrita em linguagem CES (*C for Embedded Systems*) [Thrun, 1998] [Thrun, 2000], ela deverá gerar automaticamente (através de técnicas avançadas de ILP (*Instruction-Level Parallelism*), computação reconfigurável, compilação e inteligência artificial) uma arquitetura reconfigurável para a execução desta aplicação específica e um conjunto otimizado de instruções de máquina para essa dada arquitetura. A arquitetura proposta será especificada por blocos lógicos arquiteturais escritos na linguagem de descrição de hardware Verilog [Ciletti, 1999] [Coffam, 2000] ou VHDL (*VHSIC Hardware Description Language*, onde VHSIC é a sigla de *Very High Speed Integrated Circuit*) [Ashenden, 1996] [Yalamanchili, 2001].

A proposta do ARCHITECT-R é bastante arrojada e é muito difícil ser desenvolvida por uma só pessoa. Por isso, o desenvolvimento do sistema foi distribuído entre vários alunos do LCR. Nesse contexto, esse trabalho visa o desenvolvimento de um dos módulos do ARCHITECT-R: o Gerador de Hardware Reconfigurável para Algoritmos Genéticos.

A pesquisa, além de construir um gerador de hardware, utilizou a eletrônica evolutiva como aplicação, integrando assim duas áreas que estão em ascensão no momento.

A arquitetura proposta foi especificada por blocos lógicos arquiteturais escritos na linguagem de descrição de hardware Verilog [Ciletti, 1999] [Coffam, 2000] e foi validada utilizando-se da ferramenta ModelSim [Aldec, 2002b] [Aldec, 2002c] [Aldec, 2002a].

A dissertação está organizada da seguinte maneira: o Capítulo 2 introduz os Algoritmos Genéticos (AGs) e apresenta algumas implementações tanto em hardware como em software. O Capítulo 3 mostra como é o desenvolvimento de hardware e discute a computação reconfigurável e o mostra os dispositivos existentes. O Capítulo 4 explica o que é e como funciona o sistema ARCHITECT-R. Enfim, o capítulo 5 mostra os módulos que foram desenvolvidos o

Figura 1.1: Objetivos do projeto ARCHITECT-R [Gonçalves, 2001]

capítulo 6 descreve os resultados obtidos, além da conclusão e dos trabalhos futuros.

Algoritmos Genéticos

2.1 Introdução

Nas décadas de 50 e 60 muitos cientistas da computação estudaram a evolução dos sistemas evolutivos com o intuito de utilizá-los como ferramentas para a otimização de problemas de engenharia. A idéia em todos esses sistemas foi evoluir a população de candidatos à solução de um dado problema usando operadores inspirados na variação genética e na seleção natural [Mitchell, 1996].

Os Algoritmos Genéticos (AGs) foram inventados na década de 60 por John Holland [Holland, 1975] na Universidade de Michigan e aprimorado nas décadas de 60 e 70. Em contraste com as estratégias de evolução e a programação evolutiva, o objetivo inicial de Holland não era criar um projeto de um algoritmo para resolver problemas específicos, mas sim estudar formalmente o fenômeno da adaptação que ocorre na natureza e desenvolver maneiras nas quais os mecanismos da adaptação natural pudessem ser importados para um sistema computacional. No livro *Adaptation in Natural and Artificial Systems* [Holland, 1975] Holland apresentou os Algoritmos Genéticos como uma abstração da evolução biológica e mostrou teoricamente a estrutura da adaptação de um Algoritmo Genético. O Algoritmo Genético de Holland é um

método que consiste em transformar uma população de cromossomos inicial em uma nova população usando a seleção natural juntamente com os operadores inspirados na genética: o crossover, a mutação e a inversão. O cromossomo pode ser representado por uma cadeia de zeros e uns. Cada um deles é formado de *genes*, que no caso citado seriam os bits da cadeia. Cada gene consiste de uma instância particular de *alelos*, no caso seriam os zeros e uns. O operador de seleção, utilizando-se de uma política apropriada, escolhe aquele cromossomo da população que está apto a se reproduzir. O crossover permuta subpartes de dois cromossomos, imitando rudimentarmente a recombinação biológica entre dois organismos haplóides (organismos que possuem apenas um cromossomo). A mutação troca randomicamente os valores alelos em algum lugar (bit) do cromossomo; e a inversão troca a ordem de uma seção de genes adjacentes, desta forma a ordem em que os genes se encontram é rearranjada. Aqui, como na maioria da literatura, crossover e recombinação são a mesma coisa.

Nos últimos anos, tem sido difundido entre pesquisadores, vários métodos de computação evolutiva, além de Algoritmos Genéticos, Programação Evolutiva e outras linhas semelhantes [Zebulum et al., 2002]. Hoje, normalmente os pesquisadores usam o termo Computação Evolutiva para descrever algo bem longe da original concepção de Holland.

2.2 Objetivos dos Algoritmos Genéticos

O objetivo dos Algoritmos Genéticos é a otimização de um dado problema. Um problema, dentro do conceito de Algoritmos Genéticos, apresenta os seguintes componentes [Carvalho and Lacerda., 1999]:

- Um espaço de busca, onde se encontram todas as possíveis soluções do problema;
- Uma função objetivo, que é utilizada para avaliar as soluções produzidas, associando a cada uma delas uma nota.

O primeiro passo de um Algoritmo Genético típico é a geração de uma população inicial de cromossomos extraída do domínio de soluções do problema. A cada população gerada os cromossomos são avaliados e recebem uma nota (que representa sua aptidão) que reflete a qualidade da solução que ele representa. Os indivíduos selecionados podem sofrer alterações em sua composição através dos operadores de crossover, mutação e inversão. O processo é repetido até que uma solução satisfatória seja encontrada.

Definindo-se claramente um problema a ser resolvido e dado um conjunto de cromossomos que representam as possíveis soluções do problema, um Algoritmo Genético executa os seguintes passos:

1. Uma população com n indivíduos é gerada aleatoriamente;
2. Cada indivíduo é avaliado quanto a sua aptidão;
3. Baseado nesse valor de aptidão essa população sofre um processo de seleção;
4. Os indivíduos selecionados são pareados e sofrem ou não crossover de acordo com uma probabilidade chamada de taxa de crossover;
5. Cada indivíduo sofre ou não mutação de acordo com uma probabilidade chamada de taxa de mutação;
6. Após a mutação uma nova geração é criada e o processo retorna ao passo 2;

O processo termina quando a população, ou um indivíduo gerado corresponder a uma solução satisfatória para o problema. Em termos matemáticos, a otimização consiste em achar a solução que corresponda ao ponto de máximo ou de mínimo da função objetivo, então a solução satisfatória seria um desses pontos. Cada iteração desse processo é chamado de geração. A Figura 2.1 nos mostra o que foi descrito anteriormente.

Os Algoritmos Genéticos podem ser usados em uma gama imensa de aplicações. Isso ocorre porque seus cromossomos não tem apenas uma maneira de serem representados. Um das maneiras de se representá-los é por números (binários, decimais, inteiros, reais, de ponto flutuante), outras podem ser: máquinas de estado, comandos de linguagem de programação (que podem estar em forma de árvore), etc.. Apesar dessa dissertação utilizar-se dos números binários como base, para explicação da teoria, construção e teste dos módulos, é importante sempre lembrar que o domínio de aplicação não é somente esse.

2.3 As Terminologias Biológicas

Na Biologia, a teoria da Evolução diz que o meio ambiente seleciona, em cada geração, os seres vivos mais aptos de uma população para a sobrevivência. Como resultado, apenas os escolhidos conseguem se reproduzir. Os menos adaptados geralmente são eliminados antes mesmo de gerar descendente [Darwin, 1859].

Figura 2.1: Esquema de um Algoritmo Genético simples.

Durante a reprodução ocorrem fenômenos como mutação e *crossover* (recombinação), entre outros, que atuam sobre o material genético armazenado nos cromossomos. Estes fenômenos levam à variabilidade dos seres vivos na população. Sobre essa população diversificada age a seleção natural, permitindo a sobrevivência geralmente dos seres mais adaptados.

No contexto de Algoritmos Genéticos esses termos são utilizados fazendo-se analogia a biologia real, mas as entidades que serão utilizadas nesse contexto são muito mais simples do que as utilizadas na biologia real.

- **Indivíduo:** Um simples membro da população. Nos Algoritmos Genéticos um indivíduo é formado pelo cromossomo e sua aptidão.
- **Cromossomo:** É o conjunto de genes que formam o DNA de uma célula. Toda célula

que forma um indivíduo contém o mesmo número de cromossomos. Em algoritmos genéticos, os cromossomos podem ser definidos como *strings* de DNA, ou seja, o vetor que representa o indivíduo.

- **Gene:** Blocos funcionais de DNA que codificam uma proteína em particular. Como cada proteína é responsável por um tipo de característica, isto significa que o gene é responsável por transmitir características aos indivíduos. Cada gene é localizado em um determinado local do cromossomo. Nos Algoritmos Genéticos um gene é representado por um elemento do vetor (*string*) que representa o cromossomo.
- **Genoma:** É o conjunto completo do material genético (todos os cromossomos juntos) de um organismo. Como na maioria dos Algoritmos Genéticos os organismos possuem apenas um cromossomo, pode-se dizer que genoma é o mesmo que cromossomo.
- **Genótipo:** Representação da composição genética contida no genoma. Nos Algoritmos Genéticos representa a informação contida em cada cromossomo.
- **Fenótipo:** Significa as características físicas do indivíduo, tais como cor dos olhos e cabelos, tamanho do cérebro, inteligência, etc. Nos Algoritmos Genéticos, o fenótipo representa o objeto, estrutura ou organismo construído a partir das informações do genótipo. É o cromossomo decodificado.
- **Alelo:** Representa uma das formas alternativas de um gene, ou seja, as formas que os genes podem assumir. Por exemplo, num gene que codifica a cor de olhos, os alelos são: castanho, verde, azul e preto (que são as possíveis cores dos olhos). Nos Algoritmos Genéticos, o alelo representa os valores que o gene pode assumir. Por exemplo, numa cadeia de bits os alelos seriam zero e um.
- **Epistatia:** Interação entre os genes do cromossomo, isto é, quando o valor de um gene influencia o valor de outro gene. Problemas com alta epistatia são de difíceis soluções por Algoritmos Genéticos.

2.4 Esquemas

Dentro do contexto de Algoritmos Genéticos, foi desenvolvido por Holland um conceito de Esquemas. Sua compreensão pode ajudar na construção de aplicações eficientes de Algorit-

mos Genéticos. Holland [Holland, 1975] constatou que os Algoritmos Genéticos manipulavam determinados segmentos das cadeias de bits e os chamou de Esquemas. Estes são compostos pelos símbolos 0,1 e * sendo que o * significa que nessa posição pode ocorrer 0 ou 1. A Figura 2.2 mostra alguns exemplos. O comprimento $\delta(H)$, mostrado na figura, é a diferença entre a primeira e última posição ocupada por algo diferente de *. A ordem $O(H)$ de um esquema é o número de 0's e 1's de um esquema.

	H_1	H_2	H_3
	1****	**10*	*0*01
11001	X		
11011	X		
10101	X	X	X

Comprimento $\delta(H_1) = 0$	Ordem $O(H_1) = 1$
Comprimento $\delta(H_2) = 1$	Ordem $O(H_2) = 2$
Comprimento $\delta(H_3) = 3$	Ordem $O(H_3) = 3$

Figura 2.2: Esquemas [Carvalho and Lacerda., 1999].

Todos os esquemas da Figura 2.2 H_1 , H_2 e H_3 representam cadeias de 5 bits. O $H_3 = *0*01$ representa todas as cadeias que tem 0 na segunda e quarta posições e 1 na quinta, tem $\delta(H)=3$ porque a diferença entre o primeiro e último bits fixados, segundo e quinto é de 3 e seu $O(H)=3$ porque são fixados 3 bits. Os outros esquemas são classificados analogamente.

A notação H para os esquemas é utilizada porque os esquemas definem um Hiperplano (plano de várias dimensões) em um espaço l -dimensional de uma cadeia (*string*) de bits de tamanho l . As *strings* que se encaixam no esquema são chamadas de instâncias de H [Koza, 1992].

Para exemplificar como ficam os esquemas em um hiperplano será utilizado um cubo. Existem de fato 4 modos de se enxergar esquemas em um cubo. Uma das formas de se enxergar é fazendo esquemas de $O(2)$ (ordem 2), ou seja, fixando dois bits do esquema e deixando que apenas um seja aleatório. Isso significaria associar cada esquema a dois pontos do cubo, ou seja, a uma aresta. Como o cubo tem 12 arestas, o resultado seria 12 esquemas como mostra a Figura 2.3. Outra forma seria esquemas de $O(1)$ (apenas um dos bits é fixo e dois são aleatórios). Dessa forma o resultado seriam 4 pontos do cubo, o que corresponde a uma das 6 faces. Isso é mostrado na figura 2.4. Os resultados possíveis em um domínio de 3 bits é mostrado na Tabela 2.1.

Alguns esquemas podem ser destruídos quando se realizam as operações de crossover e mutação, mesmo assim, o Teorema dos Esquemas afirma que:

Figura 2.3: Representação dos esquemas de O(2) [Koza, 1992].

Figura 2.4: Representação dos esquemas de O(1) [Koza, 1992].

Ordem do Esquema O(H)	Dimensão do Hiperplano	Representação Geométrica	Número de Pontos	Número de Esquemas
3	0	Ponto	1	8
2	1	Linha	2	12
1	2	Plano	4	6
0	3	Cubo	8	1
			Total	27

Tabela 2.1: Tabela comparativa entre a ordem do esquema e sua representação geométrica.

Pequenos esquemas contidos em bons cromossomos (i.e. aqueles com aptidão acima da média) aumentam exponencialmente nas gerações seguintes, ao passo que esquemas contidos em cromossomos ruins (i.e. aqueles com aptidão abaixo da média) tendem a desaparecer nas gerações seguintes [Goldberg, 1989].

Os bons esquemas, de pequeno tamanho, são chamados de Blocos de Construção. A informação contida nos blocos de construção é combinada com outros blocos de construção o que faz com que no decorrer das gerações criem-se cromossomos de alta aptidão. Isso denomina-se Hipótese dos Blocos de Construção.

2.5 Métodos de Seleção

Inspirados no processo de seleção natural de seres vivos criado por Darwin, os Algoritmos Genéticos selecionam os melhores cromossomos de uma população (aqueles de alta aptidão) para gerar os descendentes (que são as variantes dos pais) através dos operadores de crossover e mutação [Michalewicz, 1994]. Uma população intermediária é utilizada para alocar os cromossomos pais selecionados. Geralmente, os pais são selecionados com probabilidade proporcional a sua aptidão.

A seleção pode ser feita de muitas maneiras e cabe ao desenvolvedor escolher o que melhor se adapta a sua aplicação. Foram escolhidos, por serem os mais utilizados, dois tipos de seleção para serem detalhados. Mas será explicado brevemente cada um dos cinco principais métodos de seleção [Zebulum et al., 2002]. São eles:

- **Seleção Proporcional (Roda da Roleta):** cada indivíduo é escolhido de acordo com sua aptidão, um indivíduo com maior aptidão tem maior probabilidade de ser escolhido, esse tipo de seleção será detalhado na subseção a seguir.
- **Seleção por Torneio (Elitismo):** é escolhido o indivíduo com melhor aptidão dentre um grupo de indivíduos da população, também será detalhado a seguir.
- **Seleção por Truncamento:** apenas os t melhores indivíduos da população podem ser selecionados.
- **Seleção por Rank Linear:** os indivíduos são ordenados de acordo com sua aptidão. O indivíduo com a melhor aptidão recebe uma probabilidade η^+ de ser selecionado enquanto o pior indivíduo recebe uma probabilidade η^- de ser selecionado. Os demais indivíduos recebem uma probabilidade que é a combinação linear das probabilidades do melhor e do pior indivíduo de acordo com a sua posição na classificação.
- **Seleção por Rank Exponencial:** similar ao Rank Linear, só que ao invés de utilizar uma interpolação linear dos indivíduos ela utiliza uma interpolação exponencial.

2.5.1 Roda da Roleta

O método mais comum de seleção é o Roda Roleta. Esse método consiste em colocar todos os indivíduos distribuídos em uma roleta. Cada um ocupa um 'espaço' nessa roleta que é

proporcional à sua aptidão. Isso significa que um cromossomo que tem aptidão maior ocupará um espaço maior na roleta tendo uma chance também maior de ser escolhido [Goldberg, 1989].

A Roleta é rodada N vezes, onde N é o número de indivíduos da população. Em cada rodada um indivíduo é escolhido para agir como pai na próxima geração. Esse método pode ser implementado como segue:

1. Some o total das aptidões dos indivíduos da população (chame isso de Total);
2. Repita N vezes:
 - (a) Escolha aleatoriamente um número r entre 0 e Total;
 - (b) Some as aptidões de cada um dos cromossomos até que a soma seja maior ou igual a r ;
 - (c) Escolha esse cromossomo para a fazer parte da próxima geração.

Desta forma uma nova geração é definida.

2.5.2 Elitismo

O elitismo foi introduzido por Kenneth DeJong [DeJong, 1975] e consiste em 'eleger' o melhor indivíduo daquela população para fazer parte da próxima. Essa técnica existe para garantir que os melhores cromossomos encontrados sejam preservados. O que poderia ocorrer sem o elitismo é que um cromossomo que foi considerado melhor até agora seja modificado pelas operações de crossover e mutação ou não ser selecionado.

O elitismo pode causar significantes melhoras no desempenho dos AGs. A figura 2.5 mostra o desempenho do melhor cromossomo em cada geração, usando o AG com e sem Elitismo, com os valores plotados no gráfico representando a média de 100 execuções do AG. Observa-se que nesse problema o AG com Elitismo encontra a solução mais rápido que o sem Elitismo. Vale ressaltar que em várias execuções, o AG ocasionalmente encontra máximos locais [Michalewicz, 1994], isso torna a média dos melhores cromossomos menor que o máximo da função.

2.6 Operadores nos Algoritmos Genéticos

Pode-se definir algumas operações básicas que podem ser realizadas em Algoritmos Genéticos: Crossover e Mutação.

Figura 2.5: Desempenho do AG com e sem elitismo [Carvalho and Lacerda., 1999].

2.6.1 Crossover

O Crossover pode ser considerado um dos principais mecanismos dos Algoritmos Genéticos para explorar regiões desconhecidas no espaço de busca. O operador de Crossover cria uma variabilidade genética na população produzindo novos descendentes que são constituídos das partes de cada um dos pais. Normalmente, os pais são escolhidos segundo uma variabilidade genética; os que possuem mais aptidão são mais escolhidos do que outros.

2.6.1.1 Crossover de um ponto

O tipo mais simples de crossover é o crossover de um ponto. Esse operador escolhe aleatoriamente um local e 'troca' a subsequência que existe antes e depois daquele local entre dois cromossomos para criar dois descendentes. A operação de crossover imita rudimentarmente a recombinação biológica entre dois organismos haplóides. A figura 2.6 ilustra esse comportamento.

<i>pai₁</i>	(0010101011 100000111111)
<i>pai₂</i>	(001111110100 10010101100)
<i>filho₁</i>	(0010101011 010010101100)
<i>filho₂</i>	(00111111010 100000111111)

Figura 2.6: Crossover de um ponto.

Existe uma taxa que determina se o crossover será realizado ou não. Essa taxa é denominada

taxa de crossover e normalmente se encontra num intervalo de 60% a 90% [Carvalho and Lacerda., 1999]. Se o crossover não é aplicado, então os filhos gerados serão idênticos aos pais (isso é utilizado para que os melhores sempre permaneçam na população). Para saber se o crossover deve ser aplicado ou não, gera-se um número pseudo-aleatório no intervalo $[0,1]$. Se o número for menor do que a taxa de crossover, então o crossover se realiza.

2.6.1.2 Crossover de N pontos e Uniforme

Os outros tipos de crossover existentes são de N pontos e o uniforme. Para que se tenha um crossover de mais de um ponto é necessário que o N seja um número par. O processo consiste no seguinte: Corta-se o cromossomo em dois ou mais pontos distintos. A parte que se localiza no início e no fim de cada cromossomo se manterá e as partes que estiverem entre os cortes serão trocadas. Sempre se obedece uma ordem para que as partes sejam trocadas. Essas partes são sempre as que estão entre um corte ímpar e um corte par; ou seja, entre o primeiro e o segundo, o terceiro e o quarto e assim por diante. A figura 2.7 mostra um crossover de dois pontos e a 2.8 mostra um de quatro.

Figura 2.7: Crossover de dois pontos.

É interessante notar que o cromossomo pode ser interpretado como um anel formado pela união de suas extremidades. Observando o anel, nota-se que o crossover de 1 ponto é um caso particular do crossover de 2 pontos quando as extremidades são juntadas. Portanto, o crossover de 2 pontos desempenha também a função do crossover de 1 ponto. Este é um dos motivos pelo qual o crossover de 2 pontos é considerado melhor do que o crossover de 1 ponto. A figura 2.9 exemplifica isso.

No crossover Uniforme para cada par de pais é gerada uma máscara de bits aleatórios. Se o valor do bit for 1 então a posição correspondente a aquele bit no *pai*₁ é copiada para o *filho*₁, se for zero é copiada para o *filho*₂. No caso do *pai*₂ isso ocorre inversamente. Se o bit da máscara for 1, então o bit vai para o *filho*₂ e se for zero para o *filho*₁. A Figura 2.10 ilustra melhor o

Figura 2.8: Crossover de quatro pontos.

Figura 2.9: Crossover de dois pontos visto como um anel [Carvalho and Lacerda., 1999].

funcionamento.

Figura 2.10: Crossover Uniforme.

Existe uma relação direta entre os tipos de crossover e os esquemas. Pode-se notar claramente que em um crossover de um ponto o esquema $1^{*****}0$ será fatalmente destruído, o que não ocorre no crossover de dois pontos. Existe entretanto um porém; se o número de cortes for muito grande destrói-se com facilidade os blocos de construção.

2.6.2 Mutação

Nos Algoritmos Genéticos convencionais a mutação é benéfica para introduzir uma diversificação na população que pode estar tendendo a uma convergência prematura [Koza, 1992]. Também é comum um símbolo em particular (um alelo) desaparecer logo nos primeiros está-

gios, porque aquele alelo, em particular, está associado com um desempenho inferior. Então, devido à não linearidade dos problemas, o alelo que foi extinto pode ser exatamente aquele que era necessário para achar o desempenho ideal. A mutação, apesar de na maioria das vezes ser benéfica, faz com que a informação que existia anteriormente seja destruída. Por isso a taxa de mutação utilizada deve ser relativamente pequena (normalmente entre 0,1 a 5%).

A mutação inicia-se escolhendo aleatoriamente um ponto da cadeia de bits. Esse ponto então é invertido para formar um novo indivíduo na população. A figura 2.11 ilustra bem esse fato.

Antes	<i>filho₁</i>	(0010101010010010101100)
	<i>filho₂</i>	(0011111011100000111111)
Depois	<i>filho₁</i>	(0010 <u>0</u> 010100100101 <u>1</u> 100)
	<i>filho₂</i>	(0011111011 <u>0</u> 00000111111)

Figura 2.11: Mutação.

A mutação está ganhando uma particular atenção da comunidade de Algoritmos Genéticos no sentido de entender como os Algoritmos Genéticos podem resolver problemas complexos. Alguns estudos estão sendo feitos para comparar o crossover com a mutação. O que foi verificado é que enquanto a mutação e o crossover têm a mesma habilidade de destruir esquemas existentes o crossover é mais robusto no sentido de construir novos esquemas.

Na verdade, não há necessidade de se escolher entre usar crossover ou mutação, o que existe é uma combinação muito bem feita entre os dois utilizando um bom método de seleção.

2.7 Aplicações

Os algoritmos genéticos tem sido utilizados em um grande número de problemas científicos e de engenharia. A seguir serão apresentados alguns exemplos dessa utilização [Mitchell, 1996].

Otimização: Os AGs têm sido muito utilizados em uma grande variedade de otimizações de tarefas. Exemplo: Otimizações numéricas e problemas de otimizações: combinatoriais e de esquemas de circuitos.

Programação Automatizada: Utilizam AGs para envolver o computador na resolução de algumas tarefas e para projetar outras estruturas computacionais como celular autômato e algumas formas de redes.

Aprendizado de Máquina: Em aplicações de aprendizado de máquina, incluindo classificação e prognóstico de tarefas, como prever o comportamento de uma estrutura protéica. Os AGs também são utilizados em alguns aspectos particulares do aprendizado de máquina, tais como os tamanhos das redes neurais, e no controle do aprendizado em classificação dos sistemas e produção de sistemas simbólicos. Também é utilizado nos sensores dos robôs.

Economia: Os AGs estão desempenhando um papel importante para modelar processos de inovação, desenvolvimento de estratégias de ordenação e emergir a economia de mercado.

Sistemas Imunes: Na modelagem de vários aspectos dos sistemas naturais imunes pode-se usar AGs, incluindo mutações somáticas durante o tempo de vida e descobrimento de famílias de multi-genes durante o tempo de evolução.

Ecologia: Para modelar fenômenos ecológicos como a corrida das armas biológicas, coevolução dos hospedeiros parasitas, simbioses e o fluxo dos recursos.

Evolução e Aprendizado: Os AGs têm sido de grande utilidade no estudo de como o indivíduo aprende e como a evolução das espécies se afetam mutuamente .

Sistemas Sociais: Os AGs também podem ser usados no estudo de aspectos evolucionários da sociedade, como a evolução do comportamento social das colônias de insetos e, mais genericamente, a evolução da cooperação e comunicação em sistemas multi-agentes.

2.8 Implementações de Algoritmos Genéticos

Será descrita duas implementações de AGs, uma em software e outra em Hardware.

2.8.1 Os Algoritmos Genéticos na Robótica

Planejamento de Navegação [Sugihara and Smith, 1997] [Sugihara and Smith, 1996b] [Sugihara and Smith, 1996a] é uma importante tarefa no controle inteligente de robôs móveis autônomos. Esta tarefa é decomposta em duas partes Planejamento de caminho e Planejamento de trajetória. *Planejamento de caminho* é gerar o percurso sem colisões, em um ambiente com obstáculos e otimizá-lo respeitando alguns critérios. O algoritmo para o planejamento de caminho é dito *off-line* se os obstáculos do ambiente são estáticos e um caminho é gerado, e *on-line* se é capaz de produzir um novo caminho respondendo às alterações do ambiente. *Planejamento de trajetória* é listar o movimento do robô ao longo do caminho planejado.

Existem muitos métodos propostos para o Planejamento de Navegação do robô. Entretanto,

poucos estão sendo desenvolvidos no sentido de movimentar o robô em um ambiente desconhecido com possíveis objetos em movimento. Kazuo Sugihara e John Smith [Sugihara and Smith, 1997] [Sugihara and Smith, 1996b] propõem em vários artigos um planejamento de movimento adaptativo, ou seja, que possa modificar seu caminho enquanto ocorrerem mudanças no ambiente (p.ex. o robô reconhece um obstáculo que antes não existia). Isto tem uma aplicação imediata no plano de movimento dos veículos aquáticos autônomos [Sugihara and Smith, 1996a] que são baseados em sensores.

A seguir será apresentada uma pesquisa desenvolvida por Kazuo Sugihara e John Smith [Sugihara and Smith, 1997] [Sugihara and Smith, 1996b] que mostra como os AGs são usados para um planejamento de movimento adaptativo para robôs. Primeiramente, assume-se que o Plano de Movimento é feito em um terreno retangular e o caminho entre dois locais é aproximado por uma sequência de células adjacentes no *grid* correspondente ao terreno. Para simplificar, também assume-se que o tamanho do robô móvel é menor do que uma célula.

2.8.1.1 Definição do Problema

Entrada:

1. Um *grid* de m por n
2. Um ponto inicial s no *grid*.
3. Um ponto de destino d e
4. Um conjunto de obstáculos (uma coleção de células no *grid*) de dois tipos sólidos e perigosos.

O caminho não pode interceptar os obstáculos sólidos enquanto os perigosos podem ser interceptados, mas isso faz com que o caminho se torne mais longo. Para cada tonalidade de obstáculo pode ser associado um peso que representa o custo, ou seja, o quanto mais longo o caminho vai ficar.

Saída: Um caminho (definido como uma sequência de células adjacentes) entre s e d que não intercepta nenhum obstáculo sólido e que o total do tamanho multiplicado pelo peso de cada célula é o mínimo possível.

Para simplificar, foi assumido que $m = n$ e que a célula inicial s e a célula de destino d estão localizadas no início e fim, respectivamente, da diagonal principal do *grid*.

No espaço 2D, um caminho é dito *monótono em relação a coordenada x* (ou simplesmente *x-monótono*) se não existem linhas paralelas ao eixo y que interceptam o caminho em dois pontos distintos, isto é, a projeção do caminho no eixo x nunca decresce. Similarmente é definido o *y-monótono*.

Foi assumido, na Figura 2.12 que o caminho é *x-monótono* ou *y-monótono* (mas não necessariamente os dois). Assim, a codificação que foi utilizada para mapear o caminho foi representada como uma *string* binária. Um caminho *x-monótono* (ou *y-monótono*) pode ser representado como uma sequência de $n - 1$ pares de direção e distância. Cada par corresponde a um elemento da sequência (que pode ser uma linha ou coluna) e n corresponde ao número de linhas ou colunas. Essa codificação é mostrada na figura 2.12. Note que o caminho mostrado é *x-monótono* mas não *y-monótono*.

Figura 2.12: Representação binária do caminho [Sugihara and Smith, 1997]

O primeiro bit α indica se o caminho é *x* ou *y-monótono*. Se $\alpha = 0$, então o caminho é *x-monótono* e se $\alpha = 1$ o caminho é *y-monótono*. Os outros blocos representam a direção e a distância em cada linha ou coluna. Os dois primeiros bits de cada elemento representam a direção. Por exemplo, 00 (vertical), 01 (diagonal - para cima), 10 (horizontal) e 11 (diagonal - para baixo) no caso de $\alpha = 0$; e 00 (horizontal), 01 (diagonal - para esquerda), 10 (vertical) e 11 (diagonal - para direita) no caso de $\alpha = 1$. Os outros bits do bloco denotam a distância a ser percorrida (que é vista como um inteiro) se a direção é 00, senão esses bits são ignorados.

Foram utilizados dois tipos de operadores para a realização do experimento: crossover de 1 ponto e mutação. No exemplo eles utilizaram os seguintes dados: $n = 16$, tamanho da população igual a 30, taxa de crossover igual a 0.8 e de mutação = 0.01. N denota o número da geração e Fit o valor da aptidão. A Figura 2.13 mostra os caminhos encontrados em um terreno complicado. Note que o ponto de partida s está marcado com um círculo e o ponto de chegada d com um X. Nessa figura, as células pretas mostram os obstáculos sólidos e os cinzas os perigosos com pesos 1, 2 e 3 sendo incrementados segundo a tonalidade. A Figura 2.14 mostra um terreno muito mais difícil do que o outro em que o primeiro caminho válido foi encontrado na geração de número 53.

Figura 2.13: Exemplo de caminhos encontrados utilizando AG [Sugihara and Smith, 1997]

Figura 2.14: Exemplo de um terreno em Zig-Zag [Sugihara and Smith, 1996b]

2.8.2 Implementações em Hardware de Algoritmos Genéticos

2.8.2.1 Hardware-Based Genetic Algorithm (HGA)

HGA [Scott et al., 1995] é um sistema computacional completo composto por duas partes: o *front-end* que consiste simplesmente de um programa de interfaceamento rodando em um computador pessoal ou uma *workstation* e o *back-end* que consiste no hardware do HGA. Esse sistema funciona da seguinte maneira: O programa de interfaceamento lê os parâmetros do Algoritmo Genético interativamente do usuário e os escrevem na memória que é compartilhada com o *back-end*. Adicionalmente, o usuário especifica a função de aptidão em alguma linguagem de programação (ex. C) ou de especificação de hardware (ex. VHDL). Então o software traduz a especificação em uma configuração de hardware e programa a FPGA (*Field Programmable Gate Arrays*) que implementará a função de aptidão. Depois disso o *front-end* envia um sinal de "início" para o *back-end*. Quando o *back-end* do HGA detecta o sinal, ele executa o AG baseado nos parâmetros que estão compartilhados na memória. Quando termina, o *back-end* manda um sinal de "pronto" para o *front-end*. O *front-end* detecta o sinal e lê a população final da memória compartilhada. A população é escrita em um arquivo para que o usuário possa ver e utilizá-lo da forma que lhe for conveniente.

O HGA foi desenvolvido em VHDL [Ashenden, 1996] [Yalamanchili, 2001]. Isso permitiu que o hardware fosse especificado comportamentalmente ao invés de estruturalmente. Isso também permitiu a criação de um hardware geral (independente do parâmetro, o hardware será o mesmo), facilitando assim seu escalonamento.

2.8.2.2 Os Módulos e as Funções

O funcionamento básico do HGA que é mostrado na Fig. 2.15, é descrito a seguir:

1. Depois que todos os parâmetros foram escritos na memória compartilhada, o Módulo de Interface com a Memória (MIM) recebe um sinal de "início" do *front-end*. O MIM atua como principal controlador da HGA e se torna o único modo de interface com o mundo exterior.
2. O MIM notifica os módulos de aptidão (FM - *fitness module*), de crossover e mutação (CMM), o gerador de números pseudo-aleatórios (RNG) e o sequenciador da população (PS) para que o HGA possa começar sua execução. Cada um desses módulos requisita

Figura 2.15: Esquemático do HGA [Scott et al., 1995].

os parâmetros necessários para o MIM, que os busca no local apropriado da memória compartilhada.

3. O sequenciador da população começa o pipeline requisitando os membros da população do MIM, verifica se eles possuem a aptidão necessária e, se possuírem, passa-os para o módulo de seleção.
4. A tarefa do módulo de seleção (SM) é de receber os novos membros do PS e analisá-los até que um par de membros que tenha uma boa aptidão sejam encontrados baseados em um número aleatório. Na verdade, ele só verifica se o membro que foi passado pelo sequenciador estão dentro da aptidão esperada e se não estiverem os descarta. Em seguida passa o novo par de membros para o módulo de mutação e crossover (CMM), reinicializa-os e reinicia o processo de seleção.
5. Quando o módulo de crossover e mutação recebe o par de membros selecionados do SM, ele decide se realizará o crossover e a mutação baseados em valores recebidos do RNG. Quando pronto, os novos membros são mandados para o módulo de aptidão para avaliação.
6. O módulo de aptidão avalia os dois novos membros do CMM e escreve-os na memória. O FM também mantém alguns registros contendo o estado corrente do HGA, o qual é utilizado para determinar o término da execução do hardware.
7. Os passos acima continuam até que o FM determine que o HGA que está executando chegou ao fim. Então ele notifica o MIM do término e encerra a execução do hardware

do HGA e manda um sinal de "pronto" para o *front-end*.

Durante a fase de teste o módulo de sequenciador foi identificado como sendo o gargalo do HGA.

2.8.2.3 Comparação com um AG em software

Scott [Scott et al., 1995] testou um protótipo do HGA e a simulação em VHDL e comparou com um AG em software (SGA) executado em uma Silicon Graphics 4D/440 com quatro CPUs MIPS R3000, rodando a 33MHz. O SGA tem funcionalidade similar ao HGA.

O HGA e o SGA rodaram com uma população de tamanho $m=16$, membros da população com $n=3$ bits, e com aptidão de $f_w=4$ bits para testes no atual protótipo. Para testes no simulador os valores foram $m=32$, $n=4$, e $f_w=12$.

Para fazer a comparação o mais justo possível, o SGA executado foi otimizado durante a compilação. O HGA e o SGA iniciaram com a mesma população inicial. A única variação na execução foi o gerador de números pseudo-aleatórios.

Os resultados da comparação são mostrados na tabela 2.2.

Função de Aptidão $f(x)$	Num de gerações g	SGA Ciclos de Clock	HGA Ciclos de Clock	HGA Speedup (Ciclos)
x	10	97064	5636	17.222
x	20	168034	10622	15.819
$x + 5$	10	99825	5585	17.874
$x + 5$	20	170279	10945	15.558
$2x$ (add)	10	101019	5390	18.742
$2x$ (add)	20	170241	10659	15.972
$2x$ (mult)	10	101555	5390	18.841
$2x$ (mult)	20	170668	10659	16.012
x^2	10	334210	22892	14.599
x^2	20	574046	45019	12.751
$2x^3 + 45x^2 + 300x$	10	342806	22586	15.178
$2x^3 + 45x^2 + 300x$	20	589863	44503	13.254
$x^3 + 15x^2 + 500$	10	333701	21362	15.621
$x^3 + 15x^2 + 500$	20	579176	44317	13.069

Tabela 2.2: Performance do HGA e do SGA [Scott et al., 1995].

Como pode ser visto na tabela 2.2, a implementação em hardware é bem mais eficiente.

Conceitos Básicos

3.1 Modelagem de Hardware

A tarefa de desenvolvimento de hardware pode se tornar um tanto complexa quando se pensa no desenvolvimento de um computador completo, com todas as funções internas e ainda controladores de periféricos. É humanamente muito complicado de compreender o sistema em sua totalidade [Ashenden, 1995], fazendo com que haja a necessidade de se encontrar métodos que, com algum grau de confiança, desenvolvam componentes e sistemas de acordo com o esperado.

Tendo a especificação do sistema que se quer desenvolver a melhor forma é decompô-lo em uma coleção de componentes que, acoplados da maneira correta, realizam a função do sistema original. Cada um desses componentes pode ser decomposto, utilizando-se o mesmo processo descrito até que se obtenha um nível em que todos os componentes sejam suficientemente simples.

A vantagem desse tipo de implementação é que cada componente pode ser desenvolvido independentemente dos outros. Assim em um determinado estágio do processo o desenvolvedor só necessita se preocupar com um número pequeno de informações e fica salvo das preocupações do sistema como um todo.

No meio computacional é utilizado o termo **modelos** [Ashenden, 1995]. O modelo representa a informação que é relevante e ignora alguns detalhes irrelevantes. Isto implica que podem existir mais de uma implementação do mesmo sistema desde que diferentes informações sejam relevantes em diferentes contextos. Um tipo de modelo é feito concentrando-se a atenção em representar a função do sistema, enquanto que outro tipo, consiste em decompô-lo em subpartes.

Quando se necessita de um sistema digital, os requisitos do sistema devem ser especificados. Tendo-se uma boa especificação, já é sabido quais os resultados esperados por um módulo ou mesmo pelo sistema como um todo. Assim, é desenvolvido um conjunto de dados de entrada para teste e se a saída corresponder a esperada, o sistema funciona, senão é necessário que se façam alguns reparos.

É necessário que se tenha uma metodologia de desenvolvimento para que se chegue ao sistema que se deseja minimizando o tempo de desenvolvimento e o custo. É necessário que o sistema seja especificado de forma clara e objetiva para que os subsistemas sejam desenvolvidos corretamente e atendam os requisitos. O maior contribuidor para o excessivo custo de projeto de hardware é o tempo de revisão do sistema após pronto, corrigindo erros, e melhorando detalhes de desenvolvimento.

3.1.1 Domínios e Níveis de Modelamento

Na prática, descrições de um sistema digital podem ser classificadas em quatro níveis de abstração distintos. Esses níveis são, em ordem ascendente de abstração [Calazans, 1995]:

1. **Nível Elétrico:** onde os modelos dos componentes provêm da teoria de circuitos elétricos e eletrônicos, e/ou da física de semicondutores, i. e. transistores, resistores, capacitores, equações diferenciais, diagramas elétricos, etc;
2. **Nível Lógico:** onde o sistema digital é descrito a partir de noções elementares da teoria de circuitos digitais: portas lógicas, flip-flops, equações Booleanas, tabelas e grafos de transição, etc;
3. **Nível Arquitetural:** onde os modelos de base são os componentes mais complexos da teoria de circuitos digitais e as formas de descrevê-los: registradores, decodificadores, ULAs, multiplexadores, linguagens de transferência entre registradores, grafos de fluxo de dados e de controle, etc;

4. **Nível Sistêmico:** aqui, o sistema digital é descrito como um conjunto de algoritmos e módulos capazes de executar aqueles, que podem ou não ter um mapeamento direto para um "hardware" existente: processadores, memórias, componentes abstratos implementados no "software" básico do sistema, tais como um pacote de ponto flutuante em um sistema sem coprocessador aritmético, ou monitores de sistemas operacionais;

Além de decompor o processo de projeto segundo a quantidade de abstração das descrições, pode-se classificar as descrições segundo o tipo de informação que elas carregam. Essa classificação é decomposta em **domínios de descrição**, onde se entende por domínio o tipo de informação contida em uma dada descrição de sistema digital. Podemos identificar três domínios de descrição para sistemas digitais [Calazans, 1995]:

1. **Física ou Geométrica:** contém informação geométrica sobre os componentes/módulos e/ou sobre a disposição espacial destes no sistema a ser fabricado;
2. **Estrutural:** contém informação sobre como interconectar blocos de base de comportamento conhecido para realizar determinada funcionalidade, sem se ocupar com a disposição física destes no sistema;
3. **Comportamental ou Funcional:** carrega informação sobre o comportamento do sistema, sem se envolver em como tal comportamento pode ser obtido, seja do ponto de vista físico, seja do ponto de vista estrutural;

A combinação de níveis de abstração e domínios de descrição provê um modelo ao mesmo tempo simples e poderoso para interpretar o processo de projeto de sistemas digitais. Uma forma de representação de tal modelo bidimensional foi proposta pela primeira vez por Gajski e Kuhn em [Gajski and Kuhn, 1983], o chamado Diagrama Y, uma amostra do qual aparece na Figura 3.1. Neste diagrama, círculos concêntricos correspondem a níveis de abstração, enquanto que segmentos de reta radiais correspondem a domínios de descrição. Cada intersecção de um círculo com um segmento radial representa um tipo de descrição distinta do sistema digital.

Primeiro, o centro do diagrama corresponde conceitualmente à descrição final, ou seja, àquela que contém toda a informação necessária à fabricação do sistema digital. Ferramentas de projeto executam transformações em descrições, identificando-se então com as arestas do grafo que descreve o processo de projeto.

Figura 3.1: O Diagrama de Gajski-Kuhn, ou Diagrama Y [Calazans, 1995].

Para explicar melhor cada um dos níveis será considerado um microcontrolador de um chip único usado para controlar um instrumento de medida, que mostra os dados de entrada em um display.

No nível mais abstrato, a função do sistema deve ser descrita em termos algorítmicos, muito parecido com um algoritmo para um programa de computador. Esse nível no modelo funcional é chamado de **Modelo Comportamental**, esse termo é adotado quando se apresenta uma descrição abstrata da função do sistema.

Nesse nível de abstração, a estrutura do sistema deve ser descrita como interconexões de componentes como processadores, memórias e dispositivos de entrada e saída. Esse nível as vezes é chamado de PSM (*Processor Memory Switch*), nomeado após a notação usada por Bell e Newell [Bell and Newell, 1971]. Ele é constituído de um processador (P), uma memória (M), interconexões (S - *switchs*) e dois controladores de entrada e saída (Kio).

No domínio geométrico nesse nível de abstração, o sistema a ser implementado como um circuito VLSI *Very Large System Integration* deve ser modelado utilizando-se um *floor plan*. Isso mostra como os componentes descritos no modelo estrutural são arranjados no silício.

O segundo nível de abstração na modelagem, descrito no segundo anel da Figura 3.1 descreve o sistema em termos de unidades de armazenamento de dados e transformações. No

domínio estrutural, é normalmente chamada de RTL (*Register-Transfer Level*) composta de um *caminho de dados* e uma *seção de controle*. O caminho dos dados contém os dados gravados em registradores e os dados são transferidos entre eles através de unidades de transformação. A seção de controle coloca em sequência as operações dos componentes do caminho de dados.

No domínio funcional, o RTL é geralmente usado para especificar a operação do sistema nesse nível. A gravação dos dados é representada utilizando-se variáveis de registrador, as transformações são representadas por operadores lógicos e aritméticos.

No domínio geométrico, esse tipo de modelo depende do meio físico. Depende do tipo de chip que é utilizado e de como é a distribuição dos registradores e outros dispositivos.

O terceiro nível de abstração mostrado na Figura 3.1 é o nível de lógica convencional. Nesse nível, as estruturas são modeladas utilizando-se interconexões de portas lógicas e as funções são modeladas através de equações booleanas e tabelas verdade. No meio físico circuito integrado customizado, geometricamente deve ser modelado usando um grid (matriz) virtual.

Para que se obtenha um nível de detalhamento maior no modelamento das estruturas, deve-se usar transistores individuais, funções utilizando equações diferenciais que relatam a voltagem e a corrente no circuito. E geometricamente usando polígonos para cada máscara da camada do circuito integrado. A maioria dos desenvolvedores não necessitam trabalhar nesse nível de detalhamento, as ferramentas de desenvolvimento permitem uma tradução automática de um nível mais alto de abstração.

3.1.2 Linguagens de Modelamento

Assim como diferentes tipos de modelos podem ser usados para representar os vários níveis do arranjo funcional, estrutural e físico do sistema, também existem diferentes formas de expressar esses modelos dependendo do seu uso. Por exemplo, o modelo estrutural é comumente representado em forma de esquemático do circuito. Símbolos gráficos são utilizados para representar os subsistemas e instâncias dos mesmos são conectados utilizando-se linhas que representam os fios. Essa é a forma preferida dos desenvolvedores, no entanto, a mesma estrutura pode ser representada textualmente em forma de lista de conectores.

Quando se trata do domínio funcional, normalmente utiliza-se notação textual para o modelamento. Utilizando-se uma linguagem de especificação de hardware, o desenvolvedor pode descrever a operação do sistema sem indicar o que realmente deve ser implementado em nível de portas lógicas. Isso significa que ele estará desenvolvendo hardware, mas em um nível mais

alto de abstração.

As linguagens mais utilizadas são VHDL [Ashenden, 1996] [Yalamanchili, 2001] e Verilog [Ciletti, 1999] [Coffam, 2000]. As linguagens de descrição de hardware são normalmente baseadas em métodos de matemática formal, como lógica temporal e máquinas de estado. Além disso, possui declaração de: blocos lógicos, fios de interconexão, registradores, portas lógicas, operações lógicas. Além disso, a linguagem permite que alguns comportamentos do hardware dependam de algum evento específico, por exemplo, subida de borda do *clock*.

Baseando-se nesse contexto, foram desenvolvidos os módulos que implementam os Algoritmos Genéticos descritos no Capítulo 2.

3.2 Computação Reconfigurável

Existem classicamente duas formas de se executar um algoritmo computacional [Compton, 1999] [Compton and Hauck, 1999]: através de ASICs (*Application Specific Integrated Circuit*) ou microprocessadores. Os ASICs (também chamados de hardware totalmente customizados) são processadores projetados para executarem uma determinada operação. Logo, eles são rápidos e eficientes quando estão executando a operação para a qual eles foram desenvolvidos. Contudo, após fabricado o ASIC não pode ser alterado. Caso seja necessário modificá-lo, ele tem que ser desenvolvido e fabricado novamente, o que é uma tarefa bastante onerosa, tanto do ponto de vista do tempo despendido quanto do dinheiro gasto no processo.

Por outro lado, os microprocessadores são dispositivos bastante flexíveis que podem ser programados para desempenhar qualquer operação digital. Entretanto, ele tem um desempenho menor quando comparado com um ASIC por causa do tempo gasto na busca e decodificação de instruções e por causa da sua não-especialização. A Figura 3.2 ilustra de uma maneira bastante clara o mencionado anteriormente.

Uma nova forma de computação que vem ganhando destaque, principalmente nos últimos anos, é a computação reconfigurável [Villasenor and Mamgione-Smith, 1997] [DeHon, 1996] [DeHon and Wawrzynek, 1997] [Brown and Rose, 1996] [Compton and Hauck, 2000] [Hauck, 1998] [Compton and Hauck, 1999] [Beckett, 2000]. A computação reconfigurável vem preencher o gap existente entre o hardware e o software, possuindo potencialmente uma performance bem maior do que o software enquanto mantém a flexibilidade que inexistia no hardware. Esse tipo de computação é baseado em *Field-Programmable Gate Arrays (FPGAs)*.

Figura 3.2: Vantagens da computação reconfigurável [Jenne et al., 2002].

3.2.1 Field-Programmable Gate Arrays (FPGAs)

As FPGAs consistem em um arranjo de elementos lógicos configuráveis contidos em um único chip. Cada um desses elementos, que são chamados de blocos lógicos, contém uma capacidade computacional para implementar funções lógicas e/ou realizar roteamento permitindo a comunicação entre células. A comunicação entre os blocos é feita através dos recursos de interconexão. Dessa forma, circuitos customizados podem ser mapeados na FPGA computando-se as funções lógicas nos blocos lógicos e usando os recursos de interconexão para juntar os blocos, formando o circuito necessário. A borda externa do arranjo consiste de blocos especiais capazes de realizar operações de entrada e saída. Existem vários tipos de FPGAs e para diferenciar entre uma e outra, elas foram classificadas em famílias. A seguir, serão descritas a família Stratix que foi desenvolvida pela Altera.

As FPGAs (*Field-Programmable Gate Array*) da Altera, foram desenvolvidas tendo-se em mente que o uso futuro dos dispositivos lógicos programáveis serão os chamados SOPCs (*System on a Programmable Chip*). SOPCs são sistemas que integram toda a funcionalidade necessária (processadores, periféricos, memória, entre outras coisas) num único chip. Para atingir esse objetivo, os dispositivos por ela desenvolvidos possuem grande capacidade e diversidade de elementos lógicos internos. Além disso, conseguem acessar dispositivos externos com uma velocidade bastante elevada (via de regra, apenas dispositivos de entrada e saída para a interface com o usuário).

3.2.2 Família Stratix

A família Stratix [Altera, 2003a] [Altera, 2002c] corresponde ao estado da arte em termos de dispositivos lógico programáveis da Altera, uma das principais fabricantes de PLDs do mercado mundial. Todos os dispositivos da família stratix contém uma arquitetura baseada em um conjunto bidimensional de linhas e colunas (*array*) para implementar uma lógica customizada. Essas linhas e colunas, que possuem velocidades e tamanhos diferentes, interligam os LABs (*Logic Array Blocks*), blocos DSP (*Digital Signal Processing*) e blocos de memória. Além disso, a stratix possui dispositivos de entrada e saída (IOE - *I/O Elements*) distribuídos ao longo de toda a periferia do chip. A estrutura básica da família é mostrada na Figura 3.3. Cada um dos componentes será detalhado a seguir:

Figura 3.3: Diagrama de blocos dos FPGAs da família Stratix [Altera, 2003a].

3.2.2.1 Memória

Um importante diferencial da família stratix é a estrutura TriMatrixTM de memória desenvolvida pela Altera (Fig 3.4). Essa estrutura consiste de três tamanhos diferentes de blocos de memória RAM para implementar memórias *true dual-port* e *buffers FIFO (first-in first-out)*.

Os três tipos de blocos de memória são M512RAM, M4K e M-RAM. Os blocos **M512 RAM** são memórias *simple dual-port* de 512 bits mais paridade (576 bits) que podem ser acessadas no modo *single-port* ou *simple dual-port* com largura de até 18 bits, velocidade de até 318 MHz. Esses, são agrupados em colunas ao longo do dispositivo entre alguns dos LABs. Blocos de **M4K RAM** são *true dual-port* com 4k bits de paridade a mais (4.608 bits). Esses blocos possuem *true dual-port*, *simple dual-port*, ou *single-port memory* de largura de até 36-bits e velocidade até 291 MHz. Eles, assim como os anteriores, estão agrupados em colunas ao longo da FPGA, entre os LABs. Por último as **M-RAM** são blocos de memória *true dual-port* com 512k bits de paridade a mais (589.824 bits). Esses blocos possuem *true dual-port* dedicada, *simple dual-port*, ou *single-port* com largura de até 144 bits e velocidade de até 269 MHz. Muitos blocos de M-Ram estão localizados individualmente ou em pares no interior do array lógico.

Figura 3.4: Tecnologia TriMatrix da Altera [Altera, 2003a].

3.2.2.2 Blocos de DSP (*Digital Signal Processing*)

Blocos de DSP são elementos especiais de hardware que foram introduzidos de forma a acelerar aplicações DSP. Tais aplicações, comumente, necessitam muito de multiplicações; o que não é muito eficiente de ser implementado em FPGA. Para suprir essa deficiência, a Altera introduziu esses blocos. Cada bloco possui três modos de operação (visando as mais diferentes aplicações DSP): multiplica, multiplica e soma (ou subtrai) e multiplica e acumula (Fig. 3.5). Cada um desses três modos pode utilizar operandos de três diferentes tamanhos: 9, 18 e 36 bits. Esses blocos também contém entrada para registradores de *shift* para aplicações de processamento de sinal digital incluindo filtros FIR (*Finite Impulse Response*) e IIR (*Infinite Impulse Response*). Os blocos de DSP estão agrupados em duas colunas em cada dispositivo.

Figura 3.5: Estrutura geral dos blocos de DSP [Altera, 2003a]

A quantidade de blocos DSP, M512 RAM, M4K RAM e M-RAM variam juntamente com o número de colunas e linhas de acordo com cada dispositivo desta família, conforme a 3.2.2.2.

Dispositivos	M512 RAM Coluna/Bloco	M4K Coluna/Bloco	M-RAM Bloco	Bloco DSP Coluna/Bloco	LAB Coluna	LAB Linha
EP1S10	4 / 94	2 / 60	1	2 / 6	40	30
EP1S20	6 / 194	2 / 82	2	2 / 10	52	41
EP1S25	6 / 224	3 / 138	2	2 / 10	62	46
EP1S30	7 / 295	3 / 171	4	2 / 12	67	57
EP1S40	8 / 384	3 / 183	4	2 / 14	77	61
EP1S60	10 / 574	4 / 292	6	2 / 18	90	73
EP1S80	11 / 767	4 / 364	9	2 / 22	101	91

Tabela 3.1: Quantidade de recursos dos dispositivos da família Stratix [Altera, 2003a].

3.2.2.3 Dispositivos de Entrada e Saída

Cada pino de entrada e saída (I/O) da família Stratix é alimentado por um IOE (*I/O Element*) localizado no final das linhas e colunas dos LABs ao redor de todo o dispositivo. Os pinos de I/O suportam vários padrões de I/O. Cada IOE contém um buffer de I/O bidirecional e seis registradores para guardar as entradas, saídas e sinais *enable* de saída. Quando usado com clock dedicado, esses registradores obtêm uma performance excepcional e suporta uma interface com dispositivos de memória externa como dispositivos DDR SDRAM, FCRAM, ZBT, e

QDR SRAM. Canais de interface serial de alta velocidade suportam transferências de até 840 Mbps (Mega bits por segundo) usando tecnologias padrão de IO como: LVDS, LVPECL, 3.3-V PCML, ou HyperTransport.

3.2.2.4 *Logic Array Blocks - LABs*

Cada LAB consiste de: 10 LEs, cadeias de *carry* dos LE, controle de sinal de LAB, interconexões locais, cadeia de LUTs, e cadeia de linhas de conexão. As interconexões locais, transferem sinais entre os LEs no mesmo LAB. As conexões da cadeia de LUTs transferem as saídas de um LUT do LE para o LE adjacente, com a finalidade de melhorar as sucessivas de conexões de LUT do mesmo LAB. A cadeia de registradores transfere a saída de um registrador de um LE para o registrador de outro LE adjacente no mesmo LAB.

As interconexões locais do LAB ligam os LEs internamente no mesmo LAB. A interconexão local de cada LAB forma uma arquitetura de três lados, pois podem ser dirigidas às linhas, às colunas e às saídas dentro do mesmo LAB. Os dispositivos que rodeiam o LAB, blocos M512 RAM, M4K RAM de memória ou blocos de DSP da direita ou da esquerda, também podem dirigir as interconexões locais diretamente para o *link* de interconexão. Os atributos dos *link* de interconexão minimizam o uso das interconexões das linhas e colunas, provendo maior performance e flexibilidade. Essa arquitetura minimiza o uso das interconexões em linha e coluna, permitindo que cada LE seja ligado a outros 30 LEs. A Figura 3.6 mostra a estrutura dos LABs.

3.2.3 **Hardware Virtual**

O hardware reconfigurável introduz um novo conceito que antes não existia: o conceito de hardware virtual. O conceito de hardware virtual é o de que só é necessário ter ativo no sistema o hardware dedicado para executar as tarefas que estão sendo executadas naquele instante. Deste modo, sempre que uma parte do hardware virtual não é mais necessária ela é "eliminada" do chip e sempre que uma nova funcionalidade é requerida ela é "inserida" no chip (o que é análogo ao conceito de memória virtual). Essa característica de poder possuir hardware virtual faz com que o hardware reconfigurável se adapte as reais necessidades da aplicação (software). Com isso, obtém-se a flexibilidade do software com o desempenho do hardware. Além disso, preenche-se a lacuna que existia anteriormente entre o software (flexível e lento) e o hardware (inflexível e rápido) [Compton and Hauck, 1999] [Compton and Hauck, 2000].

O hardware reconfigurável pode ser classificado segundo dois critérios [Compton, 1999]: acoplamento e reconfigurabilidade.

Em relação ao acoplamento têm-se as seguintes possibilidades (Fig.3.7):

- **Unidades Funcionais Reconfiguráveis (RPU - *Reconfigurable Processing Unit* ou FU - *Functional Unit*):** neste modelo o hardware reconfigurável é utilizado como se fosse uma unidade funcional do processador principal, acessando os registradores do pró-

Figura 3.6: Estrutura LAB da Stratix [Altera, 2003a].

Figura 3.7: Formas de acoplamento de um hardware reconfigurável [Compton and Hauck, 1999].

prio processador para armazenar seus dados de entrada e de saída. Isto permite um ambiente de programação tradicional com a adição de algumas instruções especiais que podem mudar com o tempo.

- **Coprocessador:** neste modelo o coprocessador é inicializado pelo processador principal que diz quais são, ou onde estão, os dados que o coprocessador necessita. O coprocessador efetua as operações necessárias independente do processador principal e retorna os resultados após o término da sua execução. Esse modo implica em um *overhead* maior de comunicação entre o processador e o hardware reconfigurável. Contudo pode aumentar o grau de paralelismo do sistema visto que o coprocessador opera independente do processador. Geralmente os coprocessador reconfiguráveis são maiores do que as unidades funcionais reconfiguráveis.
- **Processador:** agora o hardware reconfigurável opera como um processador adicional num ambiente multiprocessado. O *overhead* deste modelo é ainda maior do que no modelo coprocessador. Entretanto, o hardware reconfigurável ganha mais independência do processador, o que possibilita que ainda mais partes do processamento sejam passadas para o hardware reconfigurável.
- **Standalone:** aqui o hardware reconfigurável opera sozinho, sem a interferência de um processador. Esse modelo é recomendado para casos onde deseja-se processamento de alta capacidade que necessita de um hardware dedicado.

Em relação a reconfigurabilidade tem-se os seguintes modos (Fig.3.8):

Figura 3.8: Formas de reconfiguração [Compton and Hauck, 1999].

- **Total:** implica em que todo o hardware tem que ser reconfigurado de uma só vez.
- **Parcial:** permite que apenas alguns trechos do hardware sejam reconfigurados, mantendo-se os demais trechos intactos.
- **Estática:** o dispositivo necessita ser desligado para ser reconfigurado.
- **Dinâmica:** o dispositivo pode ser reconfigurado em tempo de execução.
- **De contexto único:** apenas uma configuração é carregada por vez.
- **De múltiplos contextos:** múltiplas configurações podem estar carregadas em um determinado momento. Contudo, apenas uma configuração pode estar ativa (sendo executada). Esse modelo permite uma reconfiguração rápida do dispositivo, a qual é feita através da troca da configuração ativa pela configuração necessária.

3.2.4 Sistemas Reconfiguráveis

No início, a computação reconfigurável era utilizada apenas para resolver problemas intrinsecamente paralelos, ou seja, problemas onde a habilidade do hardware reconfigurável de explorar o paralelismo espacial [DeHon, 2000] garantia um desempenho muito maior do que a dos dispositivos de propósito geral e, em algumas vezes, até mesmo do que os supercomputadores. Contudo, essas tarefas eram bastante específicas e na maioria das vezes poderiam ser melhor solucionadas através do uso de ASICs.

A tendência moderna é a de se utilizar a computação reconfigurável para acelerar aplicações de propósito geral ou de uma área específica [Vemuri and Harr, 2000]. Para isso é necessário o desenvolvimento de novas ferramentas que apoiem o desenvolvimento das aplicações para esses sistemas. Em especial o desenvolvimento de novos compiladores que sejam capazes de tirar o maior proveito possível do hardware reconfigurável. O desenvolvimento dessas ferramentas ainda está numa fase bastante inicial, mas os resultados obtidos já são bastante promissores [Waingold et al., 2000] [Ye et al., 2000] [Goldstein et al., 2000] [Wirthlin and Hutchings, 1995] [Callahan et al., 2000] [Razdan and Smith, 1994].

3.2.5 Perspectivas da Computação Reconfigurável

Os chips de computadores estão ficando cada vez mais densos (com mais portas lógicas) e essa tendência deverá ser mantida nos próximos anos. Entretanto, espera-se que por volta de 2010

este aumento de densidade dos chips não resulte mais em um aumento da velocidade (*clock*) ou desempenho (*throughput*) dos computadores de propósito geral [Beckett, 2000]. Espera-se que nessa época os chips consigam comportar por volta de um bilhão de portas lógicas. Com isso, o objetivo dos projetistas de hardware será descobrir a melhor forma de aproveitar toda essa quantidade de recursos . Uma possível resposta para esse problema é a introdução de itens customizados nos processadores de propósito geral. A tendência atual é de que a computação reconfigurável seja utilizada para esse fim. Mas antes que isso se torne uma realidade, é necessário que a construção de hardware customizado utilizando computação reconfigurável se torne uma tarefa mais fácil, ou seja, preferência automatizada.

Descrição do Sistema ARCHITECT-R

O ARCHITECT-R [Gonçalves et al., 2000] [Gonçalves et al., 2001] [Gonçalves, 2002] [Gonçalves et al., 2003] é uma proposta de um sistema de computação reconfigurável aplicado a robótica que atua no desenvolvimento de pesquisas nas áreas de compiladores e arquitetura de computadores reconfiguráveis, visando à exploração máxima do paralelismo encontrado nas mais diversas aplicações dessa área. Para isso serão utilizadas técnicas modernas de otimização de compiladores [Muchnick, 1997] [Aho et al., 1986] [Cardoso and Vestias, 2003] [Aditya et al., 1998], exploração de paralelismo em nível de instruções (ILP) [Rau and Fischer, 1992] [Fischer and Rau, 1992] e computação reconfigurável [Villasenor and Mamgione-Smith, 1997] [Wakerly, 2000] [Mano and Kime, 2000].

O ARCHITECT-R é um projeto bastante ousado e incorpora conhecimentos de áreas bastante distintas. Portanto, é imprescindível para a sua implementação que ele seja particionado de forma a permitir que o seu desenvolvimento seja modular e incremental. Para que isso aconteça é necessário que as funcionalidades dos diferentes módulos do projeto sejam descritas de forma clara. Nas seções seguintes serão descritos os módulos do ARCHITECT-R.

4.1 Hardware do Sistema ARCHITECT-R

Uma parte do hardware do sistema ARCHITECT-R será gerado usando as ferramentas do sistema Excalibur [Altera, 2000a] da Altera. A ferramenta Excalibur da Altera é uma solução que provê tanto a capacidade de configuração dos dispositivos lógicos programáveis (PLDs) quanto o desempenho e facilidade de uso dos processadores embutidos [Altera, 2002d]. A linha Excalibur da Altera oferece *cores* de processadores RISC otimizados especialmente para lógica programável. Possuindo tanto *soft cores* como *hard cores*, esta linha é capaz de atender aos requisitos de flexibilidade e desempenho que as aplicações de robótica necessitam.

Uma outra parte do hardware do sistema ARCHITECT-R será composta por blocos arquiteturais especiais (redes neurais, algoritmos genéticos, *support vector machines*, entre outros). Esses blocos arquiteturais serão criados usando a linguagem de descrição de hardware VHDL [Ashenden, 1996] [Yalamanchili, 2001] ou Verilog [Ciletti, 1999] [Coffam, 2000], dando uma maior flexibilidade para a utilização dos mesmos. Esses blocos serão incorporados ao hardware criado pelo Excalibur na forma de Unidades Funcionais Reconfiguráveis (RPU's). Portanto, tem-se uma arquitetura de hardware bastante flexível para atender todos os requisitos dos sistemas de robótica.

4.2 Diagrama de Blocos do Sistema ARCHITECT-R

A Figura 4.1 apresenta o diagrama de blocos do sistema ARCHITECT-R. Pode-se notar que os elementos principais do ARCHITECT-R são: o *front-end compiler*¹, o *back-end compiler*² e o *LEGUS System*. Também foi introduzido no sistema um processador de I/O para o tratamento separado das instruções de entrada e saída de uma determinada aplicação. Este processador se comunicará com o hardware reconfigurável de modo a realizar as operações de carga de dados e a retirada dos resultados do processamento.

O hardware reconfigurável será inicialmente implementado com a família de FPGAs APEX [Altera, 2002d] [Altera, 2001a] [Altera, 2001b]. Para o desenvolvimento da aplicação nas FPGAs APEX será utilizado o sistema Excalibur da Altera. Tanto o hardware reconfigurável como o dispositivo de I/O serão controlados pelo kernel do sistema operacional, o qual será encarre-

¹É a parte do compilador que realiza análise léxica, sintática e semântica. Além disso também é responsável pela geração de código intermediário e pela realização de otimizações independentes de máquina.

²É a parte do compilador que realiza as otimizações dependentes de máquina e a geração de código executável.

Figura 4.1: Diagrama de blocos do ARCHITECT-R [Gonçalves, 2001].

gado, inclusive, da reconfiguração dinâmica da FPGA.

O fato do ARCHITECT-R estar usando a tecnologia de computação reconfigurável, permitirá a construção de arquiteturas que serão reconfiguradas em tempo de execução (sem que o hardware seja desligado), dando às mesmas a possibilidade de trabalhar com diversos tipos de hardware numa mesma implementação. Isto viabilizará a obtenção de um sistema mais flexível e de mais fácil adaptação a diferentes ambientes, voltado principalmente a aplicações de robótica em ambientes dinâmicos.

4.3 LEGUS System

O *LEGUS System* [Gonçalves, 2000] é formado por uma base de conhecimento, uma biblioteca de blocos arquiteturais especificados em VHDL ou Verilog e um agente inteligente que é responsável pela síntese final da arquitetura do sistema.

A idéia do *LEGUS System* é a mesma do jogo LEGO[®], onde a criança monta uma determinada estrutura escolhendo os blocos mais adequados para aquela tarefa. No *LEGUS System*, o agente inteligente imita a criança, agrupando os blocos arquiteturais que serão combinados em várias unidades funcionais e interligados com os bancos de registradores, de forma a gerar a arquitetura mais otimizada para a aplicação específica. O resultado final é um processador. Este tipo de processador é conhecido por ASIP (*Application Specific Processor*).

A base de conhecimento será composta pelas especificações dos vários blocos que podem ser usados pelo compilador para compor a arquitetura e seus atributos. A biblioteca de blocos arquiteturais possui a descrição em linguagem de descrição de hardware (HDL) dos componentes que podem ser usados na criação das RPU's. O Legus System irá operar da seguinte forma (ver Fig. 4.2):

1. Determina quais unidades funcionais reconfiguráveis podem melhorar o desempenho do sistema;
2. Determina o impacto de cada RPU escolhida no desempenho do sistema;
3. Insere as RPU's que forem possíveis, inserindo primeiro aquelas que possuem um maior impacto no desempenho do sistema;
4. Modifica o código fonte de forma a acomodar essas novas RPU's;

Figura 4.2: Diagrama de blocos do *Legus System* [Gonçalves, 2002].

5. Resolve as inconsistências geradas;
6. Permite que o ARCHITECT-R gere e simule o sistema proposto;
7. Retorna ao primeiro passo e usa os dados da simulação para corrigir possíveis erros e melhorar o desempenho;
8. Encerra o ciclo quando não for mais possível otimizar o sistema ou não conseguir resolver as inconsistências.

4.4 Considerações Finais

A ferramenta ARCHITECT-R unirá o conhecimento de várias áreas da computação e da Robótica, possibilitando uma formação de recursos humanos altamente especializados em compiladores, arquitetura de computadores, sistemas operacionais, sistemas inteligentes e computação reconfigurável. Considerando que algumas dessas áreas possuem uma grande carência de recursos humanos no país, esse projeto se torna ainda mais relevante.

Esse trabalho implementa uma RPU para Algoritmos Genéticos que será parte da biblioteca de RPUs do ARCHITECT-R. Cabe ressaltar que a linguagem CES (*C for Embedded System*) utilizada como código fonte inicial do ARCHITECT-R descreve construções utilizando-se algoritmos genéticos, as quais serão automaticamente mapeadas pela RPU em questão.

Implementação do AG

Esse capítulo apresenta a implementação do trabalho realizado. Serão descritos em detalhes a implementação bem como cada um dos módulos utilizados. Além disso, inicialmente serão fornecidas algumas informações adicionais relevantes.

5.1 Informações Adicionais

Foi utilizado uma ferramenta de simulação para validação do hardware, o ModelSim. Além disso, para tornar o código reutilizável usou-se uma metodologia de documentação chamada SRS (*Semiconductor Reuse Standard*). Essas serão detalhadas a seguir.

5.1.1 Simulador de Linguagens de Descrição de Hardware ModelSim

O ModelSim [Aldec, 2002b] [Aldec, 2002c] [Aldec, 2002a] é o simulador de linguagens de descrição de hardware muito popular. Ele possui inúmeros recursos que são indispensáveis para uma boa depuração de um hardware sendo desenvolvido, particularmente ele é bastante propício para o uso de *testbenches*¹. Essa ferramenta torna a depuração de um hardware bastante

¹Conjunto de testes especial para testar a funcionalidade do hardware

semelhante a depuração de software, simplificando o trabalho do projetista de hardware.

A principal virtude do ModelSim é poder diagnosticar de maneira mais precisa qual o erro que está acontecendo no seu hardware (caso algum esteja acontecendo) e demonstrar de maneira mais clara do que os simuladores de sinais de onda (comumente utilizados para depuração de hardware) onde está o erro.

Outra vantagem importante do simulador que ele tende a diminuir o tempo de projeto do hardware. Isso ocorre devido a maior facilidade de se compreender e encontrar os erros que podem ocorrer no hardware.

É importante salientar que todos os circuitos de hardware gerados nesse trabalho foram validados utilizando-se o ModelSim. Sem essa ferramenta, o trabalho de validação dos circuitos reconfiguráveis teria sido, com certeza, muito mais árduo.

5.2 Semiconductor Reuse Standard - SRS

Para permitir a maior reutilização possível dos blocos gerados é necessária a utilização de uma metodologia de projeto, documentação e codificação dos mesmos, que seja voltada para a reutilização. Para se atingir esse objetivo foi utilizado o *Semiconductor Reuse Standard (SRS)* [Motorola, 2001a] [Motorola, 2001d] [Motorola, 2001c] [Motorola, 2001f] [Motorola, 2001e] [Motorola, 2001b] da Motorola.

O SRS tendo sido amplamente utilizado na confecção de blocos arquiteturais ² nos centros de desenvolvimento da Motorola. O fato de que esses blocos conseguem ser reutilizados comprova a eficiência do método empregado.

A utilização do SRS como metodologia base no desenvolvimento do hardware reconfigurável implica na utilização da linguagem de descrição de hardware Verilog [Ciletti, 1999] [Coffam, 2000] e na codificação dos mesmos. No padrão SRS, existe um série de regras que devem ser seguidas com relação a nomenclatura dos módulos bem como as variáveis que serão utilizadas, como por exemplo, um *clock* deve terminar em *_clk*, um registrador deve terminar em *_reg*. Módulos da mesma instância devem ter o mesmo nome (diferenciando-se por um número: *modulo1*, *modulo2*...). Além disso, todo arquivo têm que ter um cabeçalho, onde se encontram algumas informações básicas sobre o módulo implementado no mesmo. Nesse trabalho houve uma adaptação desse padrão para o português. O cabeçalho contém informações

²A Motorola utiliza o termo componentes virtuais (VCs) e algumas outras empresas utilizam *Intellectual Property (IP)*.

úteis para a reutilização, tais como: nome do módulo, autor e e-mail do mesmo, palavras chave do módulo, propósito, parâmetros e sua descrição, etc. A Figura 5.1 mostra um exemplo do cabeçalho do modulo de crossover.

```
//+FHDR-----
// FILE NAME : crossover.v
// DEPARTMENT : LCR
// AUTHOR : Priscila Aparecida de Moraes
// AUTHOR'S EMAIL : priscila@icmc.sc.usp.br
// -----
// RELEASE HISTORY
// VERSION DATE    AUTHOR      DESCRIPTION
// 0.4    2004-01-22 Priscila A. de Moraes Updated for one child
// 0.3    2003-11-26 Priscila A. de Moraes Updated for suport crossover tax
// 0.2    2002-27-05 Priscila A. de Moraes Updated for SRS compatibility
// 0.1    2002-05-05 Priscila A. de Moraes Initial Version
// -----
// KEYWORDS : Algoritmos Genéticos, Crossover.
// -----
// PURPOSE : Realiza o crossover uniforme para uma cadeia de n bits.
// -----
// PARAMETERS
// PARAM NAME RANGE : DESCRIPTION           : DEFAULT : UNITS
// tam_cromo        [4:128] : Tamanho do cromossomo   : 16      : bits
// taxa_cross       [600:1000] : Taxa de crossover(div por 10): 600     :
// semente           : Semente (qquer num de 16 bits) : 0       :
// -----
// REUSE ISSUES
// Reset Strategy : N/A
// Clock Domains : N/A
// Critical Timing : N/A
// Test Features : N/A
// Asynchronous I/F : N/A
// Scan Methodology : N/A
// Instantiations : N/A
// Synthesizable (y/n) : Y
// Other :
// -FHDR-----
```

Figura 5.1: Cabeçalho do padrão SRS.

Essa forma de implementação do sistema possui inúmeras vantagens, onde as principais são tornar o hardware mais fácil de ser implementado e testado, permitir o uso de pipelines no sistema gerado e a fácil identificação de gargalos do sistema. Nesse trabalho, os módulos foram mantidos bem simples no início e assim que tomaram as formas desejadas foram expandidos. A seguir, será apresentada uma descrição rápida do que foi implementado e sua detalhação.

5.3 Desenvolvimento

A maneira mais comum de se implementar hardware é decompondo-o em componentes menores. Essa forma de implementação do sistema possui inúmeras vantagens, onde as principais são tornar o hardware mais fácil de ser implementado e testado, permitir o uso de pipelines no sistema gerado e a fácil identificação de gargalos do sistema. Caso um dos componentes seja identificado como sendo um gargalo do sistema, ele poderá ser decomposto em subcomponentes de granularidade mais fina, custando para isso apenas introduzir novas etapas de pipeline.

A linguagem utilizada para desenvolvimento de cada bloco foi a Verilog. Essa linguagem, como a maioria das outras utilizadas na programação de hardware, permite ao programador escrever seu código num alto nível de abstração e deixar a cargo do compilador a síntese adequada para o sistema. No entanto, os resultados obtidos podem ficar aquém do esperado. Para contornar esse problema, o ideal é que o programador faça seu código no mais baixo nível possível de abstração (se possível diretamente com portas lógicas)³ deixando pouca coisa para o compilador; garantindo assim que seu sistema tenha uma boa performance.

É importante ressaltar que todos os módulos, exceto o gerador de números aleatórios foi implementado utilizando-se parâmetros para facilitar sua reutilização.

A implementação desse trabalho ocorreu em duas etapas. Na primeira etapa foi utilizado um método de seleção (**Roda da Roleta Modificado**) que não satisfaz os requisitos do sistema. Então, foi necessária a implementação de um novo sistema, utilizando-se de outro método (**Elitismo**) para se chegar aos resultados que se esperavam. No entanto, a maioria dos módulos básicos utilizados na primeira implementação foram utilizados também na segunda. Como a primeira foi descartada, os módulos serão apenas citados e haverá uma breve descrição do seu funcionamento. Já na segunda, que foi a que atendeu os requisitos será detalhada em todos os seus módulos.

5.3.1 AG utilizando Roda da Roleta Modificado

Foram construídos diversos módulos básicos e alguns auxiliares, todos serão citados e logo após detalhados.

1. **Memória:** necessária para armazenar a população atual;

³Esse estilo já foi explicado e é conhecido por RTL (*Register Transfer Level*).

2. **Gerador de Números Aleatórios:** utilizado na seleção, crossover e mutação;
3. **Crossover:** componente que irá efetuar a operação de crossover entre os cromossomos pais;
4. **Mutação:** responsável por aplicar a mutação nos cromossomos gerados durante a etapa de crossover;
5. **Avaliador de Aptidão:** é parte do sistema que calculará a aptidão de cada cromossomo gerado.
6. **Avaliador da População:** Avalia se a população gerada é a ideal ou se há necessidade de uma nova ser gerada.

Além disso, foi necessária a criação de blocos auxiliares, tais como:

1. **Inicializador:** Inicializa com números aleatórios a população inicial;
2. **Conversor:** Recebe a cadeia de bits da memória que armazena a população atual, separa o cromossomo propriamente e alimenta o módulo de crossover;
3. **Agrupador:** Junta o novo cromossomo gerado com sua respectiva função de aptidão para poder alimentar a memória que guarda a nova população;
4. **Geradores de endereço:** Esses geram os endereços para alimentar tanto a memória que armazena os novos cromossomos gerados, quanto os endereços utilizados para a transferência entre elas.

Utilizou-se como método de seleção um roda roleta modificado. No roda roleta tradicional [Goldberg, 1989], o indivíduo de maior aptidão tem maior probabilidade de ser selecionado, nesse todos os indivíduos têm a mesma chance de ser selecionado para o processo de reprodução.

5.3.1.1 Representação

Os cromossomos utilizados são representados em forma de uma *string* binária de 16 bits. A aptidão de cada cromossomo é o número de uns que a cadeia contém. Para que a avaliação da população se tornasse viável foi necessária a criação de uma cadeia de bits que contivesse não só o cromossomo em si, mas também sua aptidão. Como a aptidão máxima é 16 (porque

o cromossomo pode ter no máximo 16 uns), então pode ser representada com 4 bits. Isso fez com que a cadeia se tornasse de 20 bits. Como a palavra das memórias são de potência de 2, a menor encontrada foi a de 32 bits. Assim a organização dos cromossomos ficou da seguinte maneira: os 12 bits mais significantes foram completados com zeros em todos os casos. Do bit 19 até o 4 localiza-se o cromossomo e do 3 ao 0 a sua nota, com mostra a Figura 5.2.

Figura 5.2: Cromossomo utilizado para a codificação dos indivíduos.

5.3.1.2 Funcionamento do Sistema

O sistema desenvolvido é descrito a seguir:

O **inicializador** gera indivíduos aleatórios e seus respectivos endereços para que sejam mandados para a memória que armazena a população atual. Após preenchida, inicia-se o processo de geração de novos indivíduos que se dá da seguinte maneira:

1. A memória manda uma a uma as cadeias de bits para o conversor;
2. O conversor separa o cromossomo e manda para o módulo de crossover, descartando o resto da cadeia;
3. O módulo de crossover recebe os dois cromossomos e uma máscara de bits randômica para a realização do crossover uniforme enviando os novos indivíduos para a mutação;
4. O módulo de mutação realiza a mutação da cadeia de bits e os envia para o Agrupador;
5. Esse agrupador concatena a cadeia com sua respectiva função de avaliação e manda para a memória que armazena a nova população).

Assim que o processo se completa e a memória que contém a nova população está completa, inicia-se o processo de transferência entre as memórias. Nesse momento, o processo de reprodução é interrompido até que a memória que guarda a população atual esteja completa. Durante esse processo a população é avaliada.

Para controlar os endereços de memória, bem como os conteúdos que serão gravados nela, são usados multiplexadores que são controlados pelo conteúdo de um registrador - processo - que tem o conteúdo 1 durante a reprodução e 0 durante a transferência.

5.3.1.3 Resultados

O sistema foi modelado e testado em um simulador de linguagens de descrição de hardware, o ModelSim [Aldec, 2002b] [Aldec, 2002c] [Aldec, 2002a]. No entanto, para que o hardware fosse mapeado foi necessária a utilização do software Quartus [Altera, 2003b]. A família de chips utilizada foi a Stratix da Altera [Altera, 2003a] e o dispositivo EP1S10F780C6ES. Os resultados obtidos são mostrados na tabela 5.1.

Família Stratix	Utilizados	Total	Porcentagem do Total
Elementos Lógicos	298	10.570	2%
Pinos	66	426	15%
Blocos DSP	0	48	0%
Bits de Memória	9.216	920.448	<1%

Tabela 5.1: Resultados obtidos no dispositivo EP1S10F780C5ES da família Stratix.

Como já mencionado, o AG não convergiu com a técnica da Roda Roleta Modificada e, portanto, os dados de experimentos não serão apresentados.

5.3.2 Utilizando Elitismo

Uma maneira encontrada para que o AG convergisse foi a utilização do método de elitismo [DeJong, 1975] para seleção. Isso já foi feito em software e foi provado que funcionou [Miazaki, 2003]. Alguns dos módulos já foram citados na implementação anterior, no entanto serão referenciados novamente para que se tenha uma noção global do sistema. Os módulos utilizados foram:

1. **Gerador de números aleatórios:** Gera a população inicial;
2. **Crossover:** Realiza o crossover dos cromossomos;
3. **Mutação:** Realiza a mutação dos cromossomos;
4. **Avaliador:** Avalia os indivíduos e guarda o melhor;
5. **Alimentador:** Gera os endereços para alimentar a memória e faz o controle do *write enable*;
6. **Memórias:** *Memória Inicial*, que guarda a população inicial e *Memória* que guarda a população corrente.

5.3.2.1 Representação

Nessa implementação não houve necessidade de guardar o *fitness* de todos os elementos da população porque só importava o melhor. Então, o melhor indivíduo e seu *fitness* foram guardados em registradores. Assim, a memória só guardava o cromossomo. Deste modo, pode ser diminuída de 32 bits para apenas 16 bits, ou seja, seu tamanho foi reduzido a metade.

5.3.2.2 Funcionamento do Sistema

Nessa implementação foi utilizado como método de seleção o elitismo. No entanto, o melhor indivíduo não só era mantido para a próxima geração, mas também recombinado com todos os outros indivíduos da população. Na primeira geração a memória inicial somente é preenchida com os indivíduos gerados aleatoriamente e os manda para o início do processo. Durante essa fase, o alimentador inicial gera os endereços e gerencia o processo de leitura e escrita. O mux seleciona se os dados são os da primeira geração (vindo da memória inicial) ou se virão da memória principal. O processo de evolução da população se dá da seguinte maneira:

1. O **Alimentador** desabilita o *write enable* da memória fazendo com que ela envie o cromossomo para o módulo de crossover;
2. O **Crossover**, recebe o indivíduo e o combina com o melhor da população (que já é conhecido assim que todos os indivíduos da população foram gerados);
3. A **Mutação** recebe os indivíduos e os muta. Além disso, manda o novo indivíduo para a memória e para o Avaliador;
4. O **Avaliador** recebe o novo indivíduo gerado, compara com o melhor até aquele momento e testa se aquela é a população ideal;
5. A **Memória** recebe e guarda os novos indivíduos;
6. O **Alimentador**, assim que percebe que existe um novo elemento na população, habilita o *write enable* fazendo com que o novo ocupe o lugar do velho. Logo após incrementa o endereço e o processo se repete.

5.3.2.3 Resultados

Assim como o sistema anterior, esse também foi modelado e testado em um simulador de linguagens de descrição de hardware, o ModelSim e mapeado utilizando-se o software Quartus. A

Família Stratix	Utilizados	Total	Porcentagem do Total
Elementos Lógicos	271	10.570	2%
Pinos	51	426	11%
Blocos DSP	0	48	0%
Bits de Memória	640	920.448	<1%

Tabela 5.2: Resultados obtidos no dispositivo EP1S10F780C5ES da família Stratix.

família de chips utilizada também foi a Stratix e o dispositivo EP1S10F780C6ES. Os resultados obtidos são mostrados na tabela 5.2.

5.3.3 Módulos Utilizados

Serão descritos brevemente o funcionamento dos módulos mais importantes, bem como os resultados obtidos em sua implementação.

5.3.3.1 Gerador de Números Aleatórios

Esse módulo é um tipo de contador com *shift*, popularmente conhecido como LFSR (*Linear Feedback Shift Register*). Um contador com *shift* usa uma pequena lógica combinacional para criar a lógica do contador. Sua lógica é muito simples, compondo-se de uma porta xor⁴ or xnor⁵ entre alguns bits específicos (dependendo do tamanho da cadeia de bits utilizada), alimentando o bit 0 e fazendo um *shift* à esquerda. Um contador LFSR pode operar em uma velocidade mais rápida que um contador binário convencional, porque a lógica envolvida é muito simples.

Esse gerador cria aleatoriamente números que variam entre 0 e $2^n - 1$, onde n é seu número de bits. Na verdade, ele gera números pseudo aleatórios. Como há necessidade de se gerar muitos números aleatórios durante o funcionamento de um AG, é fundamental que esse módulo seja rápido.

O esquema básico do gerador de números aleatórios de 16 bits é ilustrado na Figura 5.3 e descrito a seguir:

- Inicializa um registrador (Rand) com uma semente;
- A cada *clock*:
 1. Realiza-se um *shift* à esquerda nos bits de 0 a 14 (que passam a ocupar as posições de 1 a 15, fazendo com que o bit que estava na posição 15 se perca);

⁴xor: O resultado é 0 se os números são iguais e 1 se diferentes

⁵xnor: O resultado é 1 se os números são iguais e 0 se diferentes

2. O bit 0 recebe um xnor entre os bits 3, 12, 14 e 15.

- Essa operação é realizada até que retorne à semente, onde a sequência será repetida.

Figura 5.3: Gerador de Números aleatórios de 16 bits.

No exemplo mencionado anteriormente a porta utilizada para realizar a lógica é a xnor e é por esse motivo que a variável é inicializada com 0. No caso da porta ser xor, a lógica seria a mesma. A diferença é que inicializaria-se a variável com todos os bits em 1. Com esse esquema pode-se gerar números aleatórios de quantos bits forem necessários. O que muda de um gerador de 4 bits para um outro de N bits são os números que serão utilizados para realizar a operação de xnor/xor. A Tabela⁶ 5.3 nos mostra como são feitas todas as possíveis conexões para os bits variando de 3 até 32. Esse bloco não foi parametrizado, pois, como mostra a tabela, não há uma regra que relaciona os bits que deverão ser utilizados para a realização do xnor.

n	xnor entre	n	xnor entre	n	xnor entre
3	3,2	13	13,4,3,1	23	23,18
4	4,3	14	14,5,3,1	24	24,23,22,17
5	5,3	15	15,14	25	25,22
6	6,5	16	16,15,13,4	26	26,6,2,1
7	7,6	17	17,14	27	27,5,2,1
8	8,6,5,4	18	18,11	28	28,25
9	9,5	19	19,6,2,1	29	29,27
10	10,7	20	20,17	30	30,6,4,1
11	11,9	21	21,19	31	31,28
12	12,6,4,1	22	22,21	32	32,22,2,1

Tabela 5.3: Tabela das conexões do gerador de números aleatórios.[Alfke, 1996]

5.3.3.2 Resultados

A tabela 5.4 nos mostra o arquivo de saída para o gerador de números aleatórios de 4 bits. Nela verifica-se que realmente todos os números de 4 bits, exceto o 1111₂ (15₁₀), são gerados e em uma ordem aleatória.

⁶Aqui os bits começam em 1 e não em 0 como é de costume.

Número em Binário	Número em Decimal	Número em Binário	Número em Decimal
0000	0	1100	12
0001	1	1001	9
0011	3	0010	2
0111	7	0101	5
1110	14	1010	10
1101	13	0100	4
1011	11	1000	8
0110	6	0000	0

Tabela 5.4: Arquivo de Saída do teste do Bloco de Gerador de Números Aleatórios de 4 bits No arquivo, a primeira coluna corresponde aos primeiros números gerados e assim por diante..

Assim como foi feito e testado o LFSR de 4 bits, também foram feitos e testados os de 8, 16 e 32 bits. Os arquivos de saída deles, no entanto não puderam ser mostrados pois são muito grandes.

5.3.3.3 Crossover

O modo mais eficiente encontrado para gerar o filho foi o de gerar uma máscara de bits à parte. Essa será uma das entradas para o módulo. Assim, as entradas seriam: O pai1, o pai2 e a máscara. Partindo disso, seu funcionamento seria o seguinte: verifica-se o conteúdo da máscara, se o bit for 0 em uma determinada posição, então o filho recebe o conteúdo do pai1, senão (o bit da máscara é 1) o filho recebe o que está no pai2. É importante ressaltar que esse módulo gera apenas um filho. Essa solução é capaz de implementar todos os tipos de crossover existentes apenas variando a máscara de bits. Se ela for 0000000111111111, tem-se um crossover de um ponto, se for 0001111110011000, crossover de N pontos e ainda se for 0011010111010010, tem-se crossover uniforme.

Sua primeira implementação foi feita em código de alto nível deixando-se totalmente a cargo do compilador transformá-lo no hardware adequado. Após testada sua validade o sistema foi otimizado. Para sua otimização construiu-se uma tabela verdade 5.5 que tomou como base apenas um bit dos cromossomos envolvidos. Chegou-se a conclusão de que a lógica correspondente é a de um multiplexador no qual a máscara é a selecionadora dos bits.

Para essa implementação, foi utilizado o gerador de números aleatórios para a construção da máscara. Assim, o tipo de crossover implementado foi o uniforme.

Pai 1	Pai 2	Máscara	Filho
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Tabela 5.5: Tabela verdade de um bit utilizada para gerar a expressão lógica do Crossover.

5.3.3.4 Mutação

O módulo de mutação é bem simples de se implementar. Sua função é apenas a de inverter um bit do cromossomo. No entanto, existe a necessidade de se escolher qual bit será invertido e qual a probabilidade disso ocorrer. Uma possível solução encontrada foi a criação de uma máscara que indicará qual bit será mutado. Assim, se o bit da máscara for 1, o bit correspondente no cromossomo é invertido. Essa máscara, diferentemente da máscara de crossover (que poderia ter muitos 1's), tem que conter um número reduzido de 1's, porque a mutação não deve ocorrer em muitos alelos (bits) do cromossomo. Assim, foi criada uma máscara com um número fixo e reduzido de 1's. Esses 1's são embaralhados a cada ciclo de *clock*. Isso significa que um bit diferente do cromossomo será invertido a cada momento. De forma a tornar o processo mais flexível, a máscara ao invés de ter apenas n bits, tem $2n$ bits. Assim, a inicialização dos bits com 1's é proporcional a $2n$ e não a n , utilizando-se apenas os n primeiros bits. Isso faz com que o número de 1's seja variável.

Figura 5.4: Registradores da máscara da mutação.

Figura 5.5: Exemplo de conexões para a máscara se o cromossomo fosse de 4 bits.

Para gerar essa máscara foi necessária a construção de um módulo auxiliar. Este foi baseado em um registrador de *shift*, contudo não um *shift* comum, mas sim um *shift* embaralhado. Iniciam-se os bits com um número fixo de uns, dependendo do tamanho do cromossomo, e a cada *clock*, esses 1's mudam de lugar de forma embaralhada. Por exemplo, em um cromossomo de 4 bits, ao invés do bit 3 receber o bit 2, ele receberá o bit 0. Como o cromossomo é de 16 bits, então a máscara gerada é de 32 bits, de onde apenas os 16 primeiros são utilizados. A Figura 5.4 contém todos os registradores que foram utilizados e a Figura 5.5 mostra um exemplo de como os registradores seriam conectados em uma população cujos cromossomos fossem de 4 bits. Existem 8 bits, mas apenas os 4 primeiros são utilizados.

5.3.3.5 Avaliador

O Avaliador pode ser considerado o módulo mais importante nessa implementação. É ele que faz a seleção do melhor indivíduo, ou seja, daquele que será mantido para a próxima geração e realizará o crossover com todos os outros indivíduos da população.

A cada novo indivíduo gerado, o módulo realiza o seguinte processo:

1. Recebe o indivíduo;
2. Calcula sua aptidão;
3. Compara-o com a aptidão do melhor indivíduo gerado até o presente momento e, se ele for melhor, o substitui;

Esse processo se repete até que se alcance o indivíduo ou a população ideal. Para o cálculo da aptidão, o cromossomo do indivíduo é comparado bit a bit com um cromossomo META que foi projetado manualmente para navegar o robô com um bom desempenho. Se o AG conseguir evoluir os cromossomos dos indivíduos para o META nesta simulação, pode-se demonstrar que ele conseguirá configurar os controladores de navegação dos robôs reais [Simões and Dimond, 2001].

5.3.3.6 Alimentador

O Alimentador é um módulo auxiliar. Ele faz o controle dos endereços e do *write enable* da memória. Na primeira iteração ele desabilita o *write enable*, fazendo com que o dado seja colocado no barramento para sofrer os processos de crossover e mutação. Quando o novo

indivíduo está pronto, então o alimentador ativa o *write enable* para que ele substitua o antigo na memória.

Apesar de ser um módulo auxiliar, sua implementação não foi simples. Isso ocorre porque o correto preenchimento da memória que armazena a população é completamente dependente dele.

5.4 Testes

Verilog é uma linguagem que foi desenvolvida como uma linguagem de simulação e teste. Uma das principais maneiras de se testar o que foi feito é utilizando-se *testbenches*. Os *testbenches* podem ser de dois tipos: automático e não automático. O automático é aquele que existe um programa que gera os valores que serão testados e o não automático é aquele em que o projetista coloca os valores que quer testar e olha os resultados.

O *testbench* é que faz o estímulo, ou seja, ele faz com que os eventos dos quais dependem o módulo ocorram para que se possa verificar se os resultados obtidos são os esperados. Um bom exemplo, que é comumente utilizado como estímulo, é a geração de pulsos de *clock*.

Nesse trabalho, todos os módulos foram testados individualmente de forma não automática. Isso significa que para cada um deles foi criado um *testbench* específico. Foram testadas as principais funcionalidades de cada módulo para que se pudesse provar que seu funcionamento é correto.

Somente após ter certeza de que todos os módulos individualmente funcionavam corretamente é que se pode agrupá-los para formar o sistema como um todo. E além disso, para que se prove que o sistema funcione é necessário a construção de um novo *testbench*.

Todos os códigos fonte, bem como seus respectivos *testbench* estão disponíveis na seguinte url : <http://www.icmc.usp.br/~lcr/ag/> . Além disso, no site poderá ser encontrado todos os testes realizados, com suas respectivas tabelas e gráficos.

Resultados e Conclusão

6.1 Experimentos

Foram realizados testes utilizando-se da variação de parâmetros do número de indivíduos da população e das taxas de crossover e mutação. Os tamanhos das populações foram de 10, 50 e 256 indivíduos. As taxas de crossover foram de 0%, 20%, 40%, 60%, 80%, 90% e 100%, sendo que a taxa de crossover é a probabilidade que o cromossomo tem de realizar o crossover. Assim, é gerado um número entre 0 e 100. Se esse número for menor que a taxa de crossover, então o cromossomo é recombinado, caso contrário, é apenas copiado para a próxima geração. No caso das taxas de mutação, foi variado o número de 1's contido na máscara de 32 bits. Esses números foram de 1, 2, 4 e 5, que correspondem respectivamente a 4,34%, 6,65%, 12,31% e 15,23% (medidas experimentalmente). Essas porcentagens significam a probabilidade de cada bit ser mutado.

Esses testes foram realizados para 4 populações iniciais diferentes. Extraiu-se a média e calculou-se um tempo médio que a população demorou para encontrar o indivíduo ideal. Tudo foi feito em simulação e foi estipulado um cromossomo meta para ser o ideal. Esse cromossomo é um indivíduo que já se sabe que é bom e deseja-se saber quanto tempo se leva para encontrá-

lo. Pretende-se fazer pesquisas em campo utilizando-se o próprio robô para evoluir. Neste caso, deseja-se encontrar o indivíduo ideal e não se utiliza o cromossomo meta.

Os testes foram realizados e colocados em planilhas. A partir dessas planilhas foram construídos gráficos. Alguns desses gráficos serão mostrados aqui, os demais, encontram-se no Apêndice A. Baseados nesses gráficos foi construída uma tabela comparativa entre eles. Utilizando-se um FPGA da família Stratix [Altera, 2002b] [Altera, 2003a] fabricado pela Altera, os resultados obtidos são mostrados na Tabela 5.2. O tempo para a geração de cada indivíduo foi de 11,166ns. Para saber quanto tempo, em média, gasta-se para chegar no indivíduo ideal, basta multiplicar este tempo (11,166ns), o número de gerações utilizadas e o número de indivíduos. As tabelas 6.1, 6.2 e 6.3 ilustram os resultados dos testes feitos. A primeira coluna contém a variação das taxas de crossover e a primeira linha a variação das taxas de mutação. No interior, os tempos, em milisegundos, que o hardware levou para encontrar o melhor indivíduo naquelas condições.

Cross/Mut	4,344%	6,652%	12,314	15,227%
100%	2,261	2,316	3,238	5,220
90%	2,847	2,568	8,011	5,722
80%	2,289	3,014	5,160	4,522
60%	3,405	2,205	6,476	9,742
40%	3,377	4,047	5,694	7,620
20%	5,862	5,052	15,325	12,198
0%	824,385	533,697	561,342	724,812

Tabela 6.1: Tempo em ms que o hardware utiliza para encontrar o melhor indivíduo em uma população de 10 indivíduos.

Cross/Mut	4,344%	6,652%	12,314	15,227%
100%	3,210	2,232	44,664	49,549
90%	2,931	4,047	48,292	77,045
80%	3,628	3,768	54,434	123,244
60%	3,908	6,001	52,480	88,6
40%	5,164	8,234	55,132	88,630
20%	9,630	8,653	55,830	123,105
0%	773,814	2666,68	948,593	905,143

Tabela 6.2: Tempo em ms que o hardware utiliza para encontrar o melhor indivíduo em uma população de 50 indivíduos.

Pode-se notar, através das tabelas 6.1, 6.2 e 6.3, que com uma população de 256 indivíduos a convergência é rápida, isso ocorre porque são gerados muitos indivíduos a cada nova iteração e dentre eles pode estar o melhor. Por outro lado, nota-se que o pior desempenho foi encontrado

Cross/Mut	4,344%	6,652%	12,314	15,227%
100%	7,146	7,146	9,290	9,290
90%	7,146	7,146	9,290	9,290
80%	9,290	10,004	11,433	10,719
60%	9,290	10,719	10,004	15,007
40%	14,292	13,577	14,292	15,007
20%	16,198	15,007	18,580	20,724
0%	4428,579	2597,42	1637,918	2687,7

Tabela 6.3: Tempo em ms que o hardware utiliza para encontrar o melhor indivíduo em uma população de 256 indivíduos.

na população de 10 indivíduos, que no melhor caso converge em 19 gerações.

Analisando-se do ponto de vista de taxa de crossover, pode-se notar que quanto menor a taxa, mais rapidamente o hardware encontra o indivíduo ideal. Isso ocorreu para todos os tamanhos de população. Ainda com relação à taxa de crossover, um dado interessante a ser ressaltado é que a população intermediária (de 50 indivíduos) para valores baixos de crossover se assemelha mais à população de 256 indivíduos. No entanto, para taxas altas de mutação esse valor se assemelha ao da população de 10 indivíduos. Desse fato, pode-se concluir que se a taxa de crossover é alta, então, é melhor que se use uma população menor com o objetivo de cortar custos. No caso da taxa ser baixa, então o ideal é que se use uma população o quanto maior possível.

Apesar de uma população grande ser ideal, ela, como dito anteriormente, na maioria dos casos não é viável de ser implementada por falta de recursos financeiros. Então os resultados em que o trabalho se foca é o da população de 10 indivíduos.

Analisando-se as figuras, pode-se perceber pela Figura 6.1 que com uma taxa de mutação de 4,344%, a curva que encontra o melhor indivíduo primeiro é a de taxa de crossover de 100%. Em seguida vem a de 80% e logo após a de 60%. Foram essas as melhores taxas de crossover quando se usou uma taxa de mutação de 4,344%. Além disso é possível perceber que o melhor indivíduo é encontrado por volta da vigésima geração.

A Figura 6.2 apresenta os resultados de uma população de 50 indivíduos, utilizando uma taxa de mutação de 6,562% para várias taxas de crossover entre 0 e 100%. Nota-se que o melhor indivíduo foi encontrado por volta da quinta geração, ou seja, mais rapidamente do que na população de 10 indivíduos. Isso ocorre porque o espaço de busca é maior, então as chances do melhor indivíduo aparecer também é maior. Nesse caso, a melhor taxa de crossover é a de 100%. É importante observar que com uma taxa de crossover de 40%, por exemplo, leva-se em

Figura 6.1: Resultado para uma população de 10 indivíduos com 4,34% para várias taxas de crossover entre 0 e 100%.

torno de 30 gerações para encontrar o melhor indivíduo.

Figura 6.2: Resultado para uma população de 50 indivíduos com 6,562% para várias taxas de crossover entre 0 e 100%.

A partir da Figura 6.3, pode-se perceber que as curvas de taxas de crossover de 100% e 90% chegam juntas e por volta da quarta geração. É importante salientar que esse gráfico não está na mesma escala dos outros para poder melhor mostrar os detalhes. A principal diferença entre as populações de 50 e 256 indivíduos é que mesmo variando-se bastante a taxa de crossover, o melhor indivíduo é alcançado antes da décima geração.

Cabe ressaltar que todos os testes realizados, bem como seus respectivos gráficos encontram-se disponíveis na seguinte url: <http://www.icmc.usp.br/~lcr/ag/>. E que os outros gráficos encontram-se no Apêndice A.

Figura 6.3: Resultado para uma população de 256 indivíduos com 15,23% para várias taxas de crossover entre 0 e 100%.

6.2 Conclusão

O uso de AG na robótica tem se tornado cada vez mais comum e o desempenho dos mesmos está aumentando a cada dia. Uma implementação de um AG em hardware vem contribuir com a área da robótica, visto que hoje o robô necessita de um processador que execute o AG. Esse artigo apresentou uma implementação em linguagem de descrição de hardware (Verilog) de um AG que é compilada para blocos lógicos e mapeada em um FPGA, sem que se necessite de um processador para executá-lo. Em comparação com implementações em software, esta proposta apresenta maior desempenho (velocidade de execução), maior segurança (dificuldade de se copiar ou alterar a configuração do AG no chip) e robustez (implementações não processadas são em geral mais confiáveis que software rodando em processadores).

Os resultados obtidos mostram que a implementação em hardware do AG é viável e pode controlar o processo de evolução de controladores de navegação para robôs móveis autônomos com desenvoltura. Com a integração do hardware proposto com outros módulos de avaliação e supervisão do processo evolutivo, pode-se desenvolver um sistema evolutivo de controle de navegação para populações de robôs totalmente em hardware. Uma vez testado em FPGA, este sistema pode ser implementado diretamente (sem necessidade de alteração) em um ASIC (Application Specific Integrated Circuit), diminuindo ainda mais os custos de produção e aumentando a segurança e robustez do sistema.

Com relação inclusão do trabalho no projeto ARCHITECT-R, ainda não foi realizada, isso ocorre porque o ARCHITECT-R ainda está em fase de projeto, ou seja, ainda não tem um protótipo em que possam ser incluídas as RPUs.

6.3 Trabalhos Futuros

Uma das metas a ser atingida para este trabalho ser amplamente utilizado por outros membros do laboratório LCR será a sua inclusão na ferramenta *SOPC Builder* da Altera [Altera, 2002a]. A inclusão do módulo é uma tarefa complexa e envolve a formatação de todo código fonte. Entretanto uma vez anexado o módulo de AG ao *SOPC Builder*, qualquer usuário poderá usá-lo juntamente com o processador NIOS da Altera [Altera, 2000b] bastando para isso um *click* de *mouse*.

Outra importante tarefa a ser cumprida é fazer com que esse trabalho funcione no próprio robô. Então, se necessário, serão feitas algumas adaptações no hardware para que ele possa ser mapeado de forma a ser colocado no robô. Assim o robô navegará em um ambiente desconhecido e terá o objetivo de aprender sozinho a navegar sem colidir com os obstáculos.

Além disso, assim que existir um protótipo do ARCHITECT-R, essa RPU será adaptada e incluída nesse projeto.

Referências Bibliográficas

- [Aditya et al., 1998] Aditya, S., Kathail, V., and Rau, B. R. (1998). Elcor's machine description system: Version 3.0. Technical Report HPL-98-128, HP.
- [Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company.
- [Aldec, 2002a] Aldec (2002a). *ModelSim Command Reference*. Model Technology Incorporated.
- [Aldec, 2002b] Aldec (2002b). *ModelSim Tutorial*. Model Technology Incorporated.
- [Aldec, 2002c] Aldec (2002c). *ModelSim User's Guide*. Model Technology Incorporated.
- [Alfke, 1996] Alfke, P. (1996). Efficient shift registers, lfsr counters, and long pseudo-random sequence generators. Technical Report XAPP052, Xilinx.
- [Altera, 2000a] Altera (2000a). Excalibur background. white paper.
- [Altera, 2000b] Altera (2000b). Nios softcore embedded processor. data sheet.
- [Altera, 2001a] Altera (2001a). Apex 20k. data sheet.
- [Altera, 2001b] Altera (2001b). Apex 20kc. data sheet.
- [Altera, 2002a] Altera (2002a). *SOPC Builder PTF File*. Altera Corp.
- [Altera, 2002b] Altera (2002b). Stratix. Data Sheet.

- [Altera, 2002c] Altera (2002c). Using plds for high-performance dsp applications. White Paper.
- [Altera, 2002d] Altera (2002d). www.altera.com.
- [Altera, 2003a] Altera (2003a). *Stratix Device Handbook*. Altera Corp.
- [Altera, 2003b] Altera (2003b). Using quartus ii verilog hdl e vhdl integrated synthesis. Application note. an238.
- [Ashenden, 1995] Ashenden, P. J. (1995). *The designer's guide to VHDL*. Morgan Kaufman.
- [Ashenden, 1996] Ashenden, P. J. (1996). *The VHDL Designers Guide to VHDL*. Morgan Kaufmann.
- [Beckett, 2000] Beckett, J. (2000). As you like it: New technology lets non-techies customize computers. www.hpl.hp.com/news/shackleford.html.
- [Bell and Newell, 1971] Bell, C. G. and Newell, A. (1971). *Computer Structure: Readings and Examples*. McGraw-Hill, New York.
- [Brown and Rose, 1996] Brown, S. and Rose, J. (1996). Architecture of fpgas and cplds: A tutorial.
- [Calazans, 1995] Calazans, N. L. V. (1995). Métodos e ferramentas para o projeto de sistemas digitais. III ESCOLA REGIONAL DE INFORMÁTICA - ERI. Caxias do Sul - RS.
- [Callahan et al., 2000] Callahan, T. J., Hauser, J. R., and Wawrzynek, J. (2000). The garp architecture and c compiler. *IEEE Computer Magazine*.
- [Cardoso and Vestias, 2003] Cardoso, J. M. P. and Vestias, M. P. (2003). Architectures and compilers to support reconfigurable computing. www.acm.org/crossroads/xrds5-3/rconcept.html.
- [Carvalho and Lacerda., 1999] Carvalho, A. C. P. L. F. and Lacerda., E. G. M. (1999). In: Carlos O. Galvão and Mêuser J. S. Valença(org). *Sistemas Inteligentes: Aplicações a Recursos Hidricos e a Ciências Ambientais.*, chapter Algoritmos Genéticos, pages 99–150. Editora da Universidade Federal do Rio Grande do Sul.

- [Ciletti, 1999] Ciletti, M. D. (1999). *Modeling, Synthesis and Rapid Prototyping with Verilog HDL*. Prentice Hall.
- [Coffam, 2000] Coffam, K. (2000). *Real World FPGA Design with Verilog*. Prentice Hall.
- [Compton, 1999] Compton, K. (1999). Programming architectures for run-time reconfigurable systems. Master's thesis, Northwestern University.
- [Compton and Hauck, 1999] Compton, K. and Hauck, S. (1999). Configurable computing: A survey of systems and software. Technical report, University of Washington.
- [Compton and Hauck, 2000] Compton, K. and Hauck, S. (2000). An introduction to reconfigurable computing. *IEEE Computer*.
- [Darwin, 1859] Darwin, C. (1859). *On the origin of Species by means of natural selection*. London: John Murray.
- [DeHon, 1996] DeHon, A. (1996). *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT.
- [DeHon, 2000] DeHon, A. (2000). The density advantage of configurable computing. *IEEE Computer Magazine*.
- [DeHon and Wawrzynek, 1997] DeHon, A. and Wawrzynek, J. (1997). The case for reconfigurable processors. University of California at Berkeley.
- [DeHon and Wawrzynek, 1999] DeHon, A. and Wawrzynek, J. (1999). Reconfigurable computing: What, why, and design automation requirements? In *Proceedings of the 1999 Design Automation Conference*, pages 610–615.
- [DeJong, 1975] DeJong, K. (1975). *The analysis and behaviour of a class of genetic adaptative systems*. PhD thesis, University of Michigan.
- [Duckett, 2002] Duckett, T. D. (2002). *Concurrent Map Building and Self-Localisation for Mobile Robot Navigation*. PhD thesis, University of Manchester - Faculty of Science and Engineering.
- [Faraboschi et al., 2000] Faraboschi, P., Brown, G., Fisher, J. A., Desoli, G., and Homewood, F. (2000). Lx: A technology platform for customizable vliw embedded processing. *International Symposium on Computer Architecture*.

- [Fischer and Rau, 1992] Fischer, J. A. and Rau, B. R. (1992). Instruction-level parallel processing. Technical Report HPL-92-02, HP Labs.
- [Gajski and Kuhn, 1983] Gajski, D. D. and Kuhn, R. H. (1983). New vlsi tools. *IEEE Computer*, 16(12).
- [Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in search, optimization, and machine learning*. Addison-Wesley.
- [Goldstein et al., 2000] Goldstein, S. C., Schmidt, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R. R. (2000). Piperench: A reconfigurable architecture and compiler. *IEEE Computer Magazine*.
- [Gonçalves, 2000] Gonçalves, R. A. (2000). Proposta de uma ferramenta para o auxílio ao desenvolvimento de hardware. Technical report, ICMC-USP.
- [Gonçalves, 2001] Gonçalves, R. A. (2001). Architect-r: uma ferramenta para o desenvolvimento de robôs móveis. Dissertação de exame de qualificação de mestrado, ICMC-USP.
- [Gonçalves, 2002] Gonçalves, R. A. (2002). Architect-r: uma ferramenta para o desenvolvimento de robôs móveis reconfiguráveis. Master's thesis, ICMC - USP.
- [Gonçalves et al., 2003] Gonçalves, R. A., Moraes, P. A., Cardoso, J. M. P., Wolf, D. F., Fernandes, M. M., Romero, R. A. F., and Marques, E. (2003). Architect-r: A system for reconfigurable robots design in: Acm symposium on applied computing-embedded systems: Applications, solutions, and techniques. *Proceedings of the 2003 ACM Symposium on Applied Computing*. Melbourne - Florida - USA.
- [Gonçalves et al., 2000] Gonçalves, R. A., Wolf, D. F., Coelho, F. A. S., Teixeira, M. A., Ribeiro, A. A. L., and Marques, E. (2000). Architect: Um sistema de computação reconfigurável. In *CORE2000 Workshop de Computação Reconfigurável*.
- [Gonçalves et al., 2001] Gonçalves, R. A., Wolf, D. F., Rodrigues, M. I., Osorio, L. F., Moraes, P. A., Genuário, L. B., Teixeira, M. A., Ribeiro, A. A. L., Romero, R. A. F., and Marques, E. (2001). Architect-r: A system for reconfigurable robots design - an overview and initial results. *IFIP VLSI-SOC*.

- [Hauck, 1998] Hauck, S. (1998). The future of reconfigurable systems. *5th Canadian Conference on Field Programmable Devices*.
- [Hauck et al., 1997] Hauck, S., Fry, T. W., Hosler, M. M., and Kao, J. P. (1997). The chimaera reconfigurable functional unit. *IEEE Symposium on FPGAs for Custom Computing Machines*.
- [Hauser and Wawrzynek, 1997] Hauser, J. R. and Wawrzynek, J. (1997). Garp: A mips processor with a reconfigurable coprocessor.
- [Holland, 1975] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. MIT Press.
- [HP, 2002] HP (2002). Site dos laboratórios de pesquisa da hp. www.hpl.hp.com.
- [Ienne et al., 2002] Ienne, P., Pozzi, L., and Vuletic, M. (2002). Automatic processor specialisation using ad-hoc functional units.
- [Koza, 1992] Koza, J. R. (1992). *Genetic Programming of Computer by Means of Natural Selection*. MIT Press.
- [Mano and Kime, 2000] Mano, M. M. and Kime, C. R. (2000). *Logic and Computer Design Fundamentals*. Prentice Hall.
- [Miazaki, 2003] Miazaki, M. (2003). Sistema co-adaptativo para controle de robôs móveis autônomos. Projeto de Graduação.
- [Michalewicz, 1994] Michalewicz, Z. (1994). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, second edition.
- [Mitchell, 1996] Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. The MIT Press.
- [Motorola, 2001a] Motorola (2001a). Semiconductor reuse standard: Documentation. Technical Report IPMXDSRSDOC00001 V3.0, Motorola CO.
- [Motorola, 2001b] Motorola (2001b). Semiconductor reuse standard: Functional verification. Technical Report IPMXDSRSFVX00001 V3.0, Motorola CO.
- [Motorola, 2001c] Motorola (2001c). Semiconductor reuse standard: Ip interface. Technical Report IPMXDSRSIPI00001 V3.0, Motorola CO.

- [Motorola, 2001d] Motorola (2001d). Semiconductor reuse standard: Vc block deliverables. Technical Report IPMXDSRSDEL00001 V3.0, Motorola CO.
- [Motorola, 2001e] Motorola (2001e). Semiconductor reuse standard: Vc design-for-test. Technical Report IPMXDSRSDF00001 V3.0, Motorola CO.
- [Motorola, 2001f] Motorola (2001f). Semiconductor reuse standard: Verilog hdl coding. Technical Report IPMXDSRSHDL00001 V3.0, Motorola CO.
- [Muchinick, 1997] Muchinick, S. (1997). *Advanced Compiler Design and Implementation*, volume 1. Morgan Kaufmann.
- [Rau and Fischer, 1992] Rau, B. R. and Fischer, J. A. (1992). Instruction-level parallel processing: History, overview and perspective. Technical Report HPL-92-132, HP Labs.
- [Rau and Schlansker, 2001] Rau, B. R. and Schlansker, M. S. (2001). Embedded computer architecture and automation. *Computer Magazine*, pages 75–83.
- [Razdan and Smith, 1994] Razdan, R. and Smith, M. D. (1994). A high-performance microarchitecture with hardware-programmable functional units. *IEEE MICRO*.
- [Scott et al., 1995] Scott, S. D., Samal, A., and Seth, S. C. (1995). HGA: A hardware-based genetic algorithm. In *FPGA*, pages 53–59.
- [Simões, 2000] Simões, E. (2000). *Development of an embedded evolutionary controller to enable collision-free navigation of a population of autonomous mobile robots*. PhD thesis, The University of Kent at Canterbury, Canterbury - England.
- [Simões and Dimond, 2001] Simões, E. D. V. and Dimond, K. R. (2001). Embedding a distributed evolutionary system into a population of autonomous mobile robots. In *IEEE International Conference on Systems*, pages 1069–1074, Tucson-AZ - USA. Man and Cybernetics.
- [Sugihara and Smith, 1996a] Sugihara, K. and Smith, J. (1996a). Ga-based motion planning for underwater robotic vehicle. In: Proc. 10th Int’l Symp. on Unmanned Untethered Submersible Technology.
- [Sugihara and Smith, 1996b] Sugihara, K. and Smith, J. (1996b). A genetic algorithm for 3-d path planning of a mobile robot. Technical report, Universidade of Hawaii at Manoa.

- [Sugihara and Smith, 1997] Sugihara, K. and Smith, J. (1997). Genetic algorithms for adaptive motion planning of an autonomous mobile robot. Technical report, Universidade of Hawaii at Manoa.
- [Thrun, 1998] Thrun, S. (1998). A framework for programming embedded systems: Initial design and results. Technical Report CMU-CS-98-142, Carnegie Mellon.
- [Thrun, 2000] Thrun, S. (2000). Towards programming tools for robotics that integrate probabilistic computation and learning. Technical report, Carnegie Mellon.
- [Vemuri and Harr, 2000] Vemuri, R. R. and Harr, R. E. (2000). Configurable computing: Technology and applications. *IEEE Computer Magazine*.
- [Villasenor and Mamgione-Smith, 1997] Villasenor, J. and Mamgione-Smith, W. H. (1997). Configurable computing. *Scientific American*.
- [Waingold et al., 2000] Waingold, E., Taylor, M. B., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Devabhaktuni, S., Barua, R., Babb, J., Amarasinghe, S., and Agarwal, A. (2000). Baring it all to software: Raw machines. *IEEE Computer Magazine*.
- [Wakerly, 2000] Wakerly, J. F. (2000). *Digital Deseign Principles and Practices*. Prentice Hall.
- [Wirthlin and Hutchings, 1995] Wirthlin, M. J. and Hutchings, B. L. (1995). A dynamic instruction set computer. *IEEE Workshop on FPGAs for Custom Computing Machines*.
- [Yalamanchili, 2001] Yalamanchili, S. (2001). *Introductory VHDL: From Simulation to Synthesis*. Prentice Hall.
- [Ye et al., 2000] Ye, Z. A., Shenoy, N., Hauck, S., and Banerjee, P. (2000). Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th annual international symposium on Computer architecture*.
- [Zebulum et al., 2002] Zebulum, R. S., Pacheco, M. A., and Vellasco, M. M. B. R. (2002). *Evolutionary Eletronics*, volume 1 of *The CRC PRESS International Series on Computational Intelligence*. CRC PRESS, 1 edition.

Gráficos

Esse apêndice apresenta todos os gráficos que foram construídos a partir dos resultados obtidos.

Figura A.1: Resultados com População de 10 indivíduos e Mutação de 4,34%.

Figura A.2: Resultados com População de 10 indivíduos e Mutação de 6,65%.

Figura A.3: Resultados com População de 10 indivíduos e Mutação de 12,31%.

Figura A.4: Resultados com População de 10 indivíduos e Mutação de 15,23%.

Figura A.5: Resultados com População de 50 indivíduos e Mutação de 4,34%.

Figura A.6: Resultados com População de 50 indivíduos e Mutação de 6,65%.

Figura A.7: Resultados com População de 50 indivíduos e Mutação de 12,31%.

Figura A.8: Resultados com População de 50 indivíduos e Mutação de 15,23%.

Figura A.9: Resultados com População de 256 indivíduos e Mutação de 4,34%.

Figura A.10: Resultados com População de 256 indivíduos e Mutação de 6,65%.

APÊNDICE A. GRÁFICOS

Figura A.11: Resultados com População de 256 indivíduos e Mutação de 12,31%.

Figura A.12: Resultados com População de 256 indivíduos e Mutação de 15,23%.